Preliminary
Documentation

Audio and MIDI on Mac OS X

▶ **Includes "Document Revision History" (page 115)**

May 2001

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

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, ADC will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to ADC.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Audio and MIDI on Mac OS X

Welcome to audio and MIDI on Mac OS X.

Mac OS X now comes with a new audio and MIDI architecture that has been designed completely from the ground up. This new software architecture includes a comprehensive set of audio and MIDI services that are available to hardware and application developers.

If you are interested in client or application usage of these services, or in audio authoring, you should read this document.

Apple's Objectives

In creating this new architecture on Mac OS X, 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.

Developer Resources

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

- The core audio mailing list: `http://lists.apple.com/`
- The developer Web site: `http://developer.apple.com/audio`

If you are developing software support for audio hardware, you should be familiar with the Kernel and IOKit services of Mac OS X. Documentation for these services is available at

`http://developer.apple.com/techpubs/macosx/Kernel/kernel.html`

For Mac OS X development resources, refer to

`http://developer.apple.com/macosx/`

Core Audio Overview

Introduction

The audio system provided under Mac OS X presents a multi-tiered 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 the audio system 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 presented to developers 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 minimal as possible overhead in the interface of Java code to the underlying C implementation. Everything that the C interface presents to the developer can also be accomplished using Java.

Thus, if you are an application developer, you can use either or both languages, depending upon your needs and requirements. Because the Java API so closely follows the C API, understanding the overall design of these frameworks is needed as much by the Java developer as the C developer in order to effectively use the provided services.

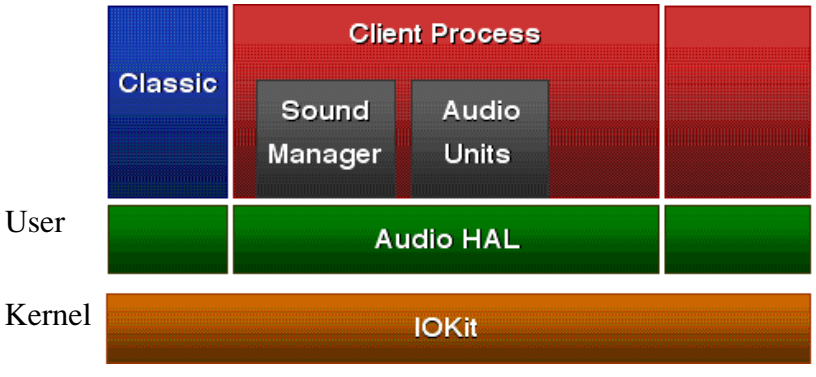
Goals

Some of the key features of the core audio architecture available on Mac OS X include

- A flexible audio format.
- Multi-channel Audio I/O.
- Support for both PCM and non-PCM formats.
- Float32 is the generic format.
- Fully-specifiable sample rates.
- Multiple application usage of Audio Devices.
- Application determined latency.
- Ubiquity of timing information.
- Both C and Java APIs.

Figure 1-1 illustrates the core audio architecture on Mac OS X and its various building blocks.

Figure 1-1 The Core Audio Architecture



The theory of operation behind the core audio architecture is discussed in subsequent chapters of this book.

The Audio Hardware Abstraction Layer (HAL)

This is presented in the `CoreAudio.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 `AudioDevice` 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 either PCM and/or 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.

Multi-channel data is represented as interleaved streams of individual samples. An AudioStream 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 AudioStream.

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; TimeStamps are ubiquitous throughout both the audio and MIDI system. This provides the capability to know the state of any particular sample (i.e., "sample accurate timing") of the device.

AudioUnits.framework

AudioUnits are 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 AudioUnit that wraps an AudioDevice), 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.

AudioUnits utilize a similar property mechanism as the CoreAudio framework and use the same structures for both the buffers of audio data and timing information. AudioUnits also provide 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 AudioUnits' rendering process.

An application can use an AudioOutputUnit to interface to a device. The DefaultOutputAudioUnit will track 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.

AudioToolbox.framework

This framework currently provides two primary services.

1. AUGraph allows for the constructions and management of a signal processing graph of AudioUnits, 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.
2. MusicSequence services provide a sequence object comprised of one or more tracks of MusicEvents (both system provided and user-defined). Track data can be edited, while a sequence is playing, and its data can be iterated over. A MusicSequence typically addresses a graph of AudioUnits, where tracks can be addressed to different nodes (AudioUnits) of its graph. A MusicPlayer is responsible for the playing of a sequence.

MIDI Services

This framework provides the representation of MIDI hardware and the inter-application 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 inter-application 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.

CHAPTER 2

The Audio Hardware Abstraction Layer (HAL)

This chapter discusses the Audio Hardware Abstraction Layer (HAL). The "Reference" (page 19) section describes the constants, data types and functions that comprise the HAL.

Overview

The Audio Hardware Abstraction Layer (HAL) represents the lowest level of access to audio devices, as well as the general characteristics of the device. It is presented in the CoreAudio.framework, as are structures and APIs that are used throughout the core audio system. The CoreAudio.framework is divided into the following groups of APIs:

- `AudioHardware.h`
- `CoreAudioTypes.h`
- `HostTime.h`

In Java, these services are defined in the `com.apple.audio.hardware` package, with the utility structures and host time services defined in `com.apple.audio.util`.

Some of its features include

- Moving data to and from the device as efficiently as possible across the User-Kernel boundary.
- Manipulating device attributes, such as volume, mute, and sample rate.
- Providing synchronization information.

Design Goals

The goals of the Audio HAL design include the following:

- A clean, low overhead signal path.
- Multiple simultaneous clients.
- Support n channels, higher bit depths and higher sample rates.
- Support solid synchronization primitives.
- A small, yet complete API.

The AudioHardware API

The Audio Hardware API provides multiple clients with simultaneous access to all audio devices attached to the host, regardless of how the connection is made, whether that connection is through PCI, USB, or Firewire. Its goal is to provide as little overhead and as clean a signal path as possible.

The APIs are presented in object-oriented C code (with Java interfaces). Because the Audio Hardware API is object-oriented, it is important to understand what the objects are and how you can manipulate them. Every object has a different set of properties, which defines the ways that you can get and send bits of data to and from the Audio Device.

The `AudioHardware.h` file provides two essential pieces of functionality for developers on Mac OS X:

- General characteristics and properties of the audio system as represented by the `AudioHardware` calls and functions. `AudioHardware` has properties the value of which can be set and retrieved. Applications can register for notifications of changes to the values of these properties.
- A particular driver's services and capabilities as expressed through its representation as an `IOAudioFamily` class presented through the `AudioDevice` type.

The Audio Device as a Unit of Encapsulation

The basis of this API is the **Audio Device**. It provides a unit of encapsulation for I/O, timing services and properties that describe and control the device. Specifically, an Audio Device represents a single I/O cycle, a clock source based on it, and all the buffers synchronized to it.

The key feature of an audio device is its ability to input and/or output audio data in some kind of describable format on a regular — i.e., time-driven — cycle. This is represented to an application through the `IOProc` of an `AudioDevice`, where an application registers with a device, either a C function or Java interface, that is called by the audio system on a predictable period. This period is determined by the size of the audio data to be processed with each call — and is a property that the application can set.

An `AudioDevice` also has properties whose values can be retrieved and set, and applications can also register for notifications of changes to the values of these properties. These properties represent "physical" properties of the device itself, i.e., they are generally not abstracted or software-emulated capabilities.

In the API, all routines and constants use the prefix, `AudioDevice`.

An Audio Device is further divided into Audio Streams. An **Audio Stream** is a set of n channels of interleaved floating point data in the case of PCM data, or a discreet and complete data stream of encoded data. It encapsulates the buffer of memory for transferring the audio data across the User-Kernel boundary. Like Audio Devices, the Audio Stream provides properties that describe and control it. Audio Streams always have a single direction, either input or output.

Audio Devices are addressed in the API by specifying whether the request is for input or output and its channel number. Channel 0 always represents the "master" channel for a device. The actual channels of the device then use a 1-based indexing scheme and are numbered consecutively up to the total number of channels for all the Audio Device's Audio Streams.

Audio Streams are addressed similarly, but omit the direction, as it is implied in the nature of the stream. The channel numbers for each Audio Stream in an Audio Device always start at 0 and are numbered consecutively up to the total number of channels in that particular Audio Stream. However, what is channel 2 for an Audio Stream will not be channel 2 for its Audio Device.

The I/O cycle of an Audio Device presents the data for all its Audio Streams, input and output, in the same call out to the client. It also provides the timestamp of when the first sample frame of the input data was acquired as well as the timestamp of when the first sample frame of the output data will be consumed by the driver. The size of the buffers used for transfer are specified per Audio Device by each process.

Format Information

Audio Streams are the "gatekeepers" of format information. Each Audio Stream on an Audio Device may have its own format, and changes to the format of one Audio Stream may affect the format of the other Audio Streams on the Audio Device.

Audio Streams can provide and consume data in any format including non-PCM encoded formats. The format properties specify the basic format of the data. It can be further specified by other properties such as the encoded description property.

Note that if an Audio Stream presents its format as linear PCM, it will always present its data as 32-bit floating point data. Any necessary conversion to the actual physical hardware format (such as 16 or 24 bit integer) are handled by the driver in order to preserve the headroom of the device's mix bus.

The format-related properties of Audio Devices simply vector the request to the stream containing the requested channel and direction.

An Audio Device should support an arbitrary number of clients, although it is not required to. An error is returned if a given device refuses to accept another client.

Properties

Most of the complexity of the API is bound up in the properties and how they work. Notifications are available for when a property's value changes.

Global Properties

Global properties of the system are prefixed with `AudioHardware`. Some of the key global properties are

- `kAudioHardwarePropertyDevices`
- `kAudioHardwarePropertyDefaultInputDevice`
- `kAudioHardwarePropertyDefaultOutputDevice`
- `kAudioHardwarePropertyRunLoop`

Getting a List of Devices — a Code Example

The code example in Listing 2-1 is important to understand because it illustrates a technique for dealing with a property whose value is a variable length list.

The list of devices which is returned in the example is an array of audio device IDs.

The code shows how you can use the `AudioHardwareGetPropertyInfo` call to get the size of the list, then allocate enough memory to hold the list, and finally get the device list itself.

The device name is a C string, so it is of variable length.

Listing 2-1 Getting a list of devices

To get the list of devices, you follow these three steps:

1. You get the size of the list.

```
UInt32theSize;
theStatus = AudioHardwareGetPropertyInfo (
                kAudioHardwarePropertyDevices,
                &theSize, NULL );

theNumberDevices = theSize / sizeof(AudioDeviceID);
```

2. You allocate enough space to hold the list.

```
theDeviceList = (AudioDeviceID*) malloc (
        theNumberDevices * sizeof(AudioDeviceID) );
```

3. You get the device list.

```
UInt32 theSize = theNumberDevices *
                    sizeof(AudioDeviceID);

theStatus = AudioHardwareGetProperty (
                kAudioHardwarePropertyDevices,
                &theSize, theDeviceList )
```

Device Properties

Audio Devices and Audio Streams have properties that describe or control some aspect of their operation, such as the current format or its name. Changes to a property's value can be scheduled to occur in real time (if the device

supports it) or can occur immediately. Clients can sign up to be notified when a property's value changes.

Every object has properties that describe and manipulate aspects of the device, such as its name, buffer size, and volume of a particular channel.

Properties are represented by a unique ID and have conventions about the kind of data they use for a value. Some properties are read-only.

The following code snippets illustrates how you can set device properties, such as the volume of a channel.

Setting Channel Volume — a Code Example

To set the volume of a channel, you follow these steps:

1. You find out if output channel 2 has a volume control.

```
BooleanisWritable;
theStatus = AudioDeviceGetPropertyInfo (
                theDeviceID, 2, false,
                kAudioDevicePropertyVolumeScalar,
                NULL, &isWritable );
```

2. If it does have a volume control, you set the value.

```
If ( (theStatus == kAudioHardwareNoError)
                && isWritable) {
    Float32 theValue;
    theStatus = AudioDeviceSetProperty (
                theDevice, NULL, 2, false,
                kAudioDevicePropertyVolumeScalar,
                sizeof(Float32), &theValue );
```

Reference

This reference section describes the constants, data types and functions that comprise the HAL available on Mac OS X.

Types

```
typedef UInt32              AudioHardwarePropertyID;
typedef void*               AudioDeviceID;
typedef UInt32              AudioDevicePropertyID;
typedef void*               AudioStreamID;

#define kAudioDeviceUnknown ((AudioDeviceID)0)
#define kAudioStreamUnknown ((AudioStreamID)0)
```

Constants

The following declarations and definitions are in `AudioHardware.h`. In Java, the specific class is notated in brackets before the supplied definitions. In the Java API, these are available in `com.apple.audio.hardware.AHConstants`.

`kAudioDevicePropertyDeviceName = 'name'`

The name of the device as a null-terminated C-string.

`kAudioDevicePropertyDeviceManufacturer= 'makr'`

The manufacturer of the device as a null-terminated C-string.

`kAudioDevicePropertyDeviceIsAlive = 'livn'`

A UInt32 where 1 means the device is installed and ready to handle requests and 0 means the device has been removed or otherwise disconnected and is about to go away completely.

After receiving notification on this property, any AudioDeviceIDs referring to the destroyed device are invalid. It is highly recommended that all clients listen for this notification.

`kAudioDevicePropertyDeviceIsRunning = 'goin'`

AUInt32 where 0 means the device is off and 1 means the device is running.

`kAudioDeviceProcessorOverload = 'over'`

This property exists so that clients can be informed when they are overloading the the I/O thread. When the HAL dectects the situation where the combined processing time exceeds the duration of the buffer, it will notify all listeners on this property. Overloading also will cause the HAL to resynch itself and restart the IO cycle to be sure that the IO thread goes to sleep. The value of this property is a UInt32, but its value has no currently defined meaning.

`kAudioDevicePropertyBufferSize = 'bsiz'`

A UInt32 containing the size of the IO buffers in bytes.

`kAudioDevicePropertyStreamConfiguration = 'slay'`

This property returns the stream configuration of the device in an AudioBufferList (with the buffer pointers set to NULL) which describes the list of streams and the number of channels in each stream. This corresponds to what will be passed into the IOProc. It is highly recommended that all clients listen for this notification.

`kAudioDevicePropertyStreamFormat = 'sfmt'`

The stream format of the stream containing the requested channel as an `AudioStreamBasicDescription`. Since formats are stream level entities, the number of channels returned with this property actually refers to the number of channels in the stream containing the requested channel. Consequently, it only gives a partial picture of the overall number of channels for the device. Use `kAudioDevicePropertyStreamConfiguration` to get the information on how the channels are divied up across the streams.

It is highly recommended that all clients listen for this notification.

`kAudioDevicePropertyStreamFormats = 'sfm#'`

An array of the AudioStreamBasicDescription's the device supports.

`kAudioDevicePropertyStreamFormatSupported = 'sfm?'`

An AudioStreamBasicDescription is passed in to query whether or not the format is supported. A `kAudioDeviceUnsupportedFormatError` will be returned if the format is not supported and `kAudioHardwareNoError` will be returned if it is supported. `AudioStreamBasicDescription` fields set to 0 will be ignored in the query, but otherwise values must match exactly.

`kAudioDevicePropertyStreamFormatMatch = 'sfmm'`

An AudioStreamBasicDescription is passed in which is modified to describe the closest match to the given format that is supported by the device. AudioStreamBasicDescription fields set to 0 should be ignored in the query and the device is free to substitute any value it sees fit. Note that the device may return a result that differs dramatically from the requested format. All matching is at the device's ultimate discretion.

`kAudioDevicePropertyVolumeScalar = 'volm'`

A Float32 between 0 and 1 that scales the volume of the device/channel across the full range of the device.

`kAudioDevicePropertyMute = 'mute'`

A UInt32 where 0 means the device is not muted and 1 means the device is muted.

`kAudioDevicePropertyPlayThru = 'thru'`

A UInt32 where 0 means play through is off and 1 means play through is on.

`kAudioHardwarePropertyDevices = 'dev#'`

An array of the AudioDeviceIDs available in the system.

`kAudioHardwarePropertyDefaultInputDevice = 'dIn '`

The AudioDeviceID of the default input device.

k`AudioHardwarePropertyDefaultOutputDevice = 'dOut'`

The AudioDeviceID of the default output device.

`kAudioHardwarePropertySleepingIsAllowed = 'slep'`

A UInt32 where 1 means this process will allow the machine to sleep and 0 will keep the machine awake. Note that the machine can still be forced to go to sleep regardless of the setting of this property.

AudioDevice

This section describes the functions that comprise the AudioDevice in Mac OS X.

AudioDeviceStart

Starts up the given IOProc.

```
AudioDeviceStart(AudioDeviceID inDevice, AudioDeviceIOProc inProc);
```

DISCUSSION

Note that the IOProc will likely get called for the first time before the call to this routine returns.

AudioDeviceStop

Stops the given IOProc.

```
AudioDeviceStop(AudioDeviceID inDevice, AudioDeviceIOProc inProc);
```

I/O Management

These routines allow a client to send and receive data on a given device. They also provide support for tracking where in a stream of data the hardware is at currently. Timestamp information is also returned.

In Java, these routines correspond to: `com.apple.audio.hardware.AudioDevice` class.

AudioDeviceAddIOProc

Installs the given IO proc for the given device. A client may have multiple IO procs for a given device.

```
AudioDeviceAddIOProc(AudioDeviceIDinDevice,
                     AudioDeviceIOProcinProc,
                     void*inClientData);
```

AudioDeviceRemoveIOProc

Removes the given IO proc for the given device.

```
AudioDeviceRemoveIOProc(AudioDeviceID inDevice, AudioDeviceIOProc
                     inProc);
```

AudioDeviceIOProc

```
(*AudioDeviceIOProc)(AudioDeviceIDinDevice,
                     const AudioTimeStamp*inNow,
                     const AudioBufferList*inInputData,
                     const AudioTimeStamp*inInputTime,
                     AudioBufferList*outOutputData,
                     const AudioTimeStamp*inOutputTime,
                     void*inClientData);

typedef OSStatus
```

DISCUSSION

This is a client-supplied routine that the hardware calls to perform an I/O transaction for a given device. All input and output is presented to the client simultaneously for processing. The `inNow` parameter is the sample time that should be used as the basis of now rather than what might be provided by a query to the device's clock. This is necessary because time will continue to

advance while this routine is executing, making the retrieval of the current time from the appropriate parameter unreliable for synch operations. The time stamp for the `inputData` represents when the data was recorded. For the output, the timestamp represents when the first sample will be played. In all cases, each timestamp is accompanied by its mapping into host time.

The format of the actual data depends of the sample format of the device as specified by its properties. It may be raw or compressed, interleaved or not interleaved as determined by the requirements of the device and its settings.

If the data for either the input or the output is invalid, the timestamp will have a value of 0. This occurs when a device does not have any inputs or outputs.

Time Management

Time operations are only valid while the device in question is running. Otherwise, a `kAudioHardwareNotRunningError` is returned.

AudioDeviceGetCurrentTime

Retrieves the current time.

```
AudioDeviceGetCurrentTime(AudioDeviceID inDevice, AudioTimeStamp*
                    outTime);
```

AudioDeviceTranslateTime

Translates the given time.

```
AudioDeviceTranslateTime(AudioDeviceIDinDevice,
                    const AudioTimeStamp*inTime,
                    AudioTimeStamp*outTime);
```

DISCUSSION

The output time formats are requested using the flags in the outTime argument. A device may or may not be able to satisfy all requests so be sure to check the flags again on output.

The key point is that it takes an input timestamp and an output timestamp. The timestamp structures have a flags field that specifies what part. The timestamp structure has six or seven different representations of time in it. You need to set the flags to say which ones are actually valid. In the output, you need to set the flags for the one in the translation in the output. On input, you specify the sample time, while on output you want the hosttime.

Device Property Management

Devices are comprised of two sections: input and output. Each section is further divided into a number of channels. A channel, which is the smallest addressable unit of a device, represents a single channel of input or output for the device. It may be an entire stream or a channel within a stream.

When getting and setting a device's properties, it is necessary to always specify exactly which part of the device to interrogate. The section is specified with a boolean argument (generally called `isInput`) where `true` refers to the input section and `false` refers to the output section. The channel is specified with an unsigned integer argument (generally called inChannel) where 0 means the master channel and greater than zero refers to the Nth indexed channel starting with index 1.

When you specify a property, you specify not just the name of the property but a boolean value, the input or output side of the device, and also a channel number.

AudioDeviceGetPropertyInfo

Retrieves information about the given property on the given channel.

```
AudioDeviceGetPropertyInfo(AudioDeviceIDinDevice,
                    UInt32inChannel,
                    BooleanisInput,
```

```
                    AudioDevicePropertyIDinPropertyID,
                    UInt32*outSize,
                    Boolean*outWritable);
```

DISCUSSION

The outSize argument will return the size in bytes of the current value of the property. The outWritable argument will return whether or not the property in question can be changed.

AudioDeviceGetProperty

Retrieves the indicated property data for the given device.

```
AudioDeviceGetProperty(AudioDeviceIDinDevice,
                    UInt32inChannel,
                    BooleanisInput,
                    AudioDevicePropertyIDinPropertyID,
                    UInt32*ioPropertyDataSize,
                    void*outPropertyData);
```

DISCUSSION

A property is specified as an ID and a channel number. The channel number allows for access to properties on the channel level. On input, ioDataSize has the size of the data pointed to by `outPropertyData`. On output, it will contain the amount written. If `outPropertydata` is NULL and `ioPropertyDataSize` is not, the amount that would have been written will be reported.

AudioDeviceSetProperty

Sets the indicated property data for the given device.

```
AudioDeviceSetProperty(AudioDeviceIDinDevice,
                    const AudioTimeStamp*inWhen,
                    UInt32inChannel,
```

```
                    BooleanisInput,
                    AudioDevicePropertyIDinPropertyID,
                    UInt32inPropertyDataSize,
                    const void*inPropertyData);
```

AudioDevicePropertyListenerProc

This routine is called when a property's value changes.

```
(*AudioDevicePropertyListenerProc)(AudioDeviceIDinDevice,
                    UInt32inChannel,
                    BooleanisInput,
                    AudioDevicePropertyIDinPropertyID,
                    void*inClientData);

typedef OSStatus
```

AudioDeviceAddPropertyListener

Sets up a routine that gets called when the property of a device is changed.

```
AudioDeviceAddPropertyListener(AudioDeviceIDinDevice,
                    UInt32inChannel,
                    BooleanisInput,
                    AudioDevicePropertyIDinPropertyID,
                    AudioDevicePropertyListenerProcinProc,
                    void*inClientData);
```

AudioDeviceRemovePropertyListener

Removes the given notification.

```
AudioDeviceRemovePropertyListener(AudioDeviceIDinDevice,
                    UInt32inChannel,
                    BooleanisInput,
                    AudioDevicePropertyIDinPropertyID,
                    AudioDevicePropertyListenerProcinProc);
```

AudioHardwareGetPropertyInfo

Retrieves information about the given property.

```
AudioHardwareGetPropertyInfo(AudioHardwarePropertyID inPropertyID,
                    UInt32*outSize,
                    Boolean*outWritable);
```

The outSize argument will return the size in bytes of the current value of the property. The outWritable argument will return whether or not the property in question can be changed.

AudioHardwareGetProperty

Retrieves the indicated property data.

```
AudioHardwareGetProperty(AudioHardwarePropertyIDinPropertyID,
                    UInt32*ioPropertyDataSize,
                    void*outPropertyData);
```

DISCUSSION

On input, ioDataSize has the size of the data pointed to by outPropertyData. On output, it will contain the amount written. If outPropertydata is NULL and

ioPropertyDataSize is not, the amount that would have been written will be reported.

AudioHardwareSetProperty

Sets the indicated property data. Global properties, by definition, don't directly affect real time, so they don't need a timestamp.

```
AudioHardwareSetProperty(AudioHardwarePropertyIDinPropertyID,
                    UInt32inPropertyDataSize,
                    void*inPropertyData);
```

AudioHardwarePropertyListenerProc

This routine is called when a property's value changes.

```
(*AudioHardwarePropertyListenerProc)(AudioHardwarePropertyIDinPropertyID,
                    void*inClientData);
```

AudioHardwareAddPropertyListener

Sets up a routine that gets called when a property is changed.

```
AudioHardwareAddPropertyListener(AudioHardwarePropertyIDinPropertyID,
                    AudioHardwarePropertyListenerProcinProc,
                    void*inClientData);
```

AudioHardwareRemovePropertyListener

Removes the given notification.

```
AudioHardwareRemovePropertyListener(AudioHardwarePropertyIDinPropertyID,
                    AudioHardwarePropertyListenerProcinProc);
```

Errors

```
enum
{
    kAudioHardwareNoError               = 0,
    kAudioHardwareNotRunningError       = 'stop',
    kAudioHardwareUnspecifiedError      = 'what',
    kAudioHardwareUnknownPropertyError  = 'who?',
    kAudioDeviceUnsupportedFormatError  = '!dat',
    kAudioHardwareBadPropertySizeError  = '!siz',
    kAudioHardwareIllegalOperationError = 'nope'
};
```

CHAPTER 3

AudioUnits

This chapter discusses the AudioUnits framework available on Mac OS X. The section "Reference" (page 37) describes the constants, data types and functions that comprise the AudioUnits framework.

Overview

In the Mac OS X audio system, AudioUnits serve a number of purposes. **AudioUnits** 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`.

The Audio Unit Framework

The `AudioUnit.framework` provides a set of services that developers can take advantage of in their own applications by using AudioUnits. The framework also provides services for those who want to develop their own AudioUnits. The framework is divided into the following groups of APIs:

- `AudioUnit.h`
- `MusicDevice.h`
- `AudioOutputUnit.h`

AudioUnits 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, AudioUnits 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 detailed information about components, refer to

`http://gemma.apple.com/techpubs/mac/MoreToolbox/MoreToolbox-333.html`

The AudioUnit API

The `AudioUnit.h` API presents the basic `AudioUnit` interface, as well as the constants that define the `AudioUnitType` (`'aunt'`), the generic sub-types (`MusicDevice` — `'musd'`, `Effects` — `'efct'`, etc), and the specific ID of an `AudioUnit`.

The specific ID of an AudioUnit represents the specific functionality of the audio unit itself. For example, the DLS Music Device is an `AudioUnit` that is able to use both Downloadable Sounds (DLS) and SoundFont 2 (SF2) files as sample data for sample-based synthesis. Its type is `'aunt'` — an `AudioUnit`. Its sub-type is `'musd'` — a music device. Its ID is `'dls '` — a DLS music device (note the space at the end).

Often a sub-type may present additional APIs to the base API presented by an AudioUnit, though this is not always true. The ID will typically present the final discriminating identifier for a particular audio unit.

For more specific information about components, refer to *Inside Macintosh: More Macintosh Toolbox*, Chapter 6, Component Manager, which is available at

`http://developer.apple.com/techpubs/mac/MoreToolbox/MoreToolbox-333.html`

Key Points

There are two key points here:

- The type, sub-type and ID correspond to the `type`, `subType` and `manufacturerID` in the `ComponentDescription` structure.
- Apple reserves the right to specify component types, sub-type, and ID fields as all lowercase characters. For instance, in the example of the DLS Music Device above, if a third-party developer implemented its own software

synthesizer, using custom algorithms, the component would require the following two values:

```
Type == 'aunt'
Sub-Type == 'musd'
```

Those values specify to the system that this component implements the Audio Unit and the extended Music Device interfaces. The component would then use the ID field to uniquely identify itself and by convention would include at least one uppercase character:

```
ID == 'AdSy'
```

where the unit was an additive synthesizer, for example.

Audio Unit State

The basic AudioUnit states are closed, open, and initialized, which correspond to these calls:

- `OpenAComponent(...)`
- `CloseComponent(...)`
- `AudioUnitInitialize(...)`
- `AudioUnitUninitialize(...)`

No significant resource allocations are expected to occur when the AudioUnit component is first opened with OpenAComponent(). AudioUnitInitialize() is called after optional configuration has occured. This is where the AudioUnit allocates and is prepared to render.

An AudioOutputUnit also may be in a "running" state. You use the `AudioUnitStart()` call to start such a unit.

`AudioUnitReset()` may be called on any initialized `AudioUnit()`. The `AudioUnitReset()` call clears any buffers, resets filter memory, and stops any playing notes (for example, in a MusicDevice software synthesizer). It places the AudioUnit back to its initialized state.

AudioUnit Sources and Destinations

AudioUnits have sources and destinations. An AudioUnit can be just a source unit, such as software synthesizers, which are presented as a type of AudioUnit defined as a MusicDevice.

An AudioUnit can also be just a destination that is attached to a hardware output device.

Some AudioUnits 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.

AudioUnit Properties

Properties represent a general and extensible mechanism for passing information to and from AudioUnits. Information is communicated via a void* data parameter and a data byte-size parameter. The type of information is identified by an `AudioUnitPropertyID`.

The AudioUnit property APIs are similar to the MIDI and AudioHAL property APIs.

Information is addressed to a particular section of an AudioUnit 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. `AudioUnitPropertyIDs` are defined in `AudioUnit/AudioUnitProperties.h` header file along with their associated data formats.

AudioUnit 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

AudioUnit 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 whole stream of N-interleaved audio (or side-band) channels.

Data can be supplied to an AudioUnit through one of two mechanisms:

1. Connecting an AudioUnit output to another AudioUnit 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 AudioUnit through the supplied callback.

The AUGraph API provides a higher-level connection service, freeing the client from calling the AudioUnit directly.

The "Pull" I/O Model

As mentioned, AudioUnits 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 whole stream of N interleaved audio channels. Connections between units are also managed via properties. Data is requested from an AudioUnit through its `AudioUnitRenderSlice` function being called by one of its destinations, or the `RenderSliceCallback` being called.

Key points about `AudioUnitRenderSlice()` arguments:

- 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.
- The `AudioBuffer` argument passes in and receives back a buffer of audio. The client may pass in a buffer 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 `inActionFlags` 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 to by the `mData` member of the `ioData` argument. In this case, `mData` 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 AudioUnit).

If the `ioData` `mData` member is NULL, then rendering may set `mData` to a buffer owned by the AudioUnit.

In any case, on return, `mData` points to the rendered audio.

The `inTimeStamp` parameter gives the AudioUnit 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 AudioUnit. Rendering is performed separately for each of its outputs. The AudioUnit 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.

The MusicDevice API

The `MusicDevice.h` file contains the extended interface for the `MusicDevice` component. In Component Manager terms, the `MusicDevice` implements the component selectors for the `AudioUnit`, as well as the additional selectors specifically for the `MusicDevice`.

This device presents an API targeted specifically toward software synthesis. The APIs present two primary means of controlling the `MusicDevice`:

- A set of APIs based around the MIDI protocol.
- An extended control protocol to provide an additional degree of specification and control of music events.

The AudioOutputUnit API

The APIs provided in `AudioOutputUnit.h` are particularly important when using `AUGraph` services, which use the start and stop methods of these kinds of units.

The essential and defining characteristic of an output unit is that it drives the processing work of its connected units. Thus, in the case of the HAL-based units, the IOProc of the attached Audio Device is responsible for driving the production of audio data. Each I/O cycle, the HAL Output Unit's RenderSlice method is called by the IOProc of the device, which then causes the render slice function of each of its inputs to be called, and so on.

Once completed, the unit places the resulting data into the output buffer of the device.

Currently, there are two types of output units shipped by Apple:

- `kAudioUnitID_HALOutput`. This type talks to an AudioDevice as specified by the user of the unit.
- `kAudioUnitID_DefaultOutput`. This is a specialized HAL output unit that provides additional services, such as sample rate conversion, and will also track the device that the user sets from the Sound Control Panel as the default device for sound output. As a result, an application can talk directly to this default output and do no further work to maintain the integrity of the audio output destination.

One could imagine other types of output units — for example, an output unit that writes data to a file, rather than an AudioDevice. In that case, a thread would be started (and stopped), and each I/O cycle would write the resulting data to that file.

Reference

This reference section describes the constants, data types and functions that comprise the AudioUnits framework available on Mac OS X.

Types

Output Device AudioUnits

```
kAudioUnitProperty_GetMicroseconds(Int32* pointing to microseconds value)

struct AudioUnitConnection
{
    AudioUnit   sourceAudioUnit;
    UInt32      sourceOutputNumber;
    UInt32      destInputNumber;
};
```

The following struct defines a callback function which renders audio into an input of an AudioUnit.

```
struct AudioUnitInputCallback
{
    AudioUnitRenderCallback   inputProc;
    void *                    inputProcRefCon;
};

struct AudioUnitParameterInfo
{
    char                  name[64];
    AudioUnitParameterUnit unit;              // unit type (Hertz,
                                              //Decibels, etc. -- see enum
)
    Float32               minValue;           // minimum legal value
    Float32               maxValue;           // maximum legal value
    Float32               defaultValue;       // initial value when
                                              // AudioUnit is first
                                              // initialized or reset
    UInt32                flags;              // read-only attributes,
etc.
};
```

Constants

```
enum {
    kAudioUnitComponentType     = FOUR_CHAR_CODE('aunt'),
    kAudioUnitSubType_Output    = FOUR_CHAR_CODE('out '),
    kAudioUnitID_SoundManagerOutput = FOUR_CHAR_CODE('smgr'),
    kAudioUnitID_HALOutput      = FOUR_CHAR_CODE('ahal'),
    kAudioUnitID_DefaultOutput  = FOUR_CHAR_CODE('def '),
    kAudioUnitSubType_MusicDevice = FOUR_CHAR_CODE('musd'),
    kAudioUnitID_DLSSynth       = FOUR_CHAR_CODE('dls '),
    kAudioUnitSubType_Encoder   = FOUR_CHAR_CODE('aenc'),
    kAudioUnitSubType_Decoder   = FOUR_CHAR_CODE('adec'),
    kAudioUnitSubType_BitDepthConverter = FOUR_CHAR_CODE('bdcv'),
    kAudioUnitSubType_SampleRateConverter = FOUR_CHAR_CODE('srcv'),
    kAudioUnitID_PolyphaseSRC   = FOUR_CHAR_CODE('poly'),
    kAudioUnitSubType_FormatConverter = FOUR_CHAR_CODE('fmtc'),
    kAudioUnitID_Interleaver    = FOUR_CHAR_CODE('inlv'),
    kAudioUnitID_Deinterleaver  = FOUR_CHAR_CODE('dnlv'),
    kAudioUnitSubType_Effect    = FOUR_CHAR_CODE('efct'),
    kAudioUnitID_MatrixReverb   = FOUR_CHAR_CODE('mrev'),
    kAudioUnitID_Delay          = FOUR_CHAR_CODE('dely'),
    kAudioUnitID_LowPassFilter  = FOUR_CHAR_CODE('lpas'),
    kAudioUnitID_PeakLimiter    = FOUR_CHAR_CODE('lmtr'),
    kAudioUnitSubType_Mixer     = FOUR_CHAR_CODE('mixr'),
    kAudioUnitID_StereoMixer    = FOUR_CHAR_CODE('smxr')
};
```

Render Flags

```
enum {
    kAudioUnitRenderAction_Accumulate = (1 << 0),
    kAudioUnitRenderAction_UseProvidedBuffer = (1 << 1),
    kAudioUnitRenderAction_PreRender = (1 << 2),
    kAudioUnitRenderAction_PostRender = (1 << 3)
};

typedef UInt32  AudioUnitRenderActionFlags;
```

Properties

```
enum {
    kAudioUnitScope_Global = 0,
    kAudioUnitScope_Input  = 1,
    kAudioUnitScope_Output = 2,
    kAudioUnitScope_Group  = 3
};

typedef UInt32 AudioUnitPropertyID;
typedef UInt32 AudioUnitParameterID;
typedef UInt32 AudioUnitScope;
typedef UInt32 AudioUnitElement;
```

Property Constants for AudioUnits

Note that Apple Computer, Inc. reserves property values from 0 -> 63999. Developers are free to use property IDs above this range at their own discretion.

```
enum
{
// Applicable to all AudioUnits in general  (0 -> 999)
    kAudioUnitProperty_ClassInfo                = 0,
    kAudioUnitProperty_MakeConnection           = 1,
    kAudioUnitProperty_SampleRate               = 2,// value is Float64
    kAudioUnitProperty_ParameterList            = 3,
    kAudioUnitProperty_ParameterInfo            = 4,
    kAudioUnitProperty_FastDispatch             = 5,
    kAudioUnitProperty_AUGraphCPULoad           = 6, //value Float32
(0->1) -> AUGraph uses this to tell AU what the current CPU load of a
unit's graph is
    kAudioUnitProperty_SetInputCallback         = 7, // value is
AudioUnitInputCallback; scope is input, element number is the input
number

// Applicable to MusicDevices               (1000 -> 1999)
    kMusicDeviceProperty_InstrumentCount        = 1000,
    kMusicDeviceProperty_InstrumentName         = 1001,
    kMusicDeviceProperty_GroupOutputBus         = 1002,
    kMusicDeviceProperty_SoundBankFSSpec        = 1003,
    kMusicDeviceProperty_InstrumentNumber       = 1004,
```

```
// Applicable to "output" AudioUnits        (2000 -> 2999)
    kAudioOutputUnitProperty_CurrentDevice      = 2000
                    // value is AudioDeviceID
                    // will work for HAL and default output components
};
```

General AudioUnit Properties

Unless otherwise stated, assume that the `inScope` parameter is `kAudioUnitScope_Global` and the `inElement` parameter is ignored.

`kAudioUnitProperty_ClassInfo(void* points to AudioUnit-defined internal state)`

`kAudioUnitProperty_MakeConnection(AudioUnitConnection*)`

Pass in `kAudioUnitScope_Input` for the `AudioUnitScope`.

Pass in the input number for `AudioUnitElement` (redundantly stored in AudioUnitConnection).

`kAudioUnitProperty_SampleRate(Float64*)`

`kAudioUnitProperty_ParameterInfo(AudioUnitParameterInfo*)`

Pass in for the AudioUnitElement.

MusicDevice Properties

`kMusicDeviceProperty_InstrumentCount(UInt32* pointing to count )`

`kMusicDeviceProperty_InstrumentName(formatted as char*)`

Pass in `MusicDeviceInstrumentID` for the AudioUnitElement.

`kMusicDeviceProperty_GroupOutputBus(UInt32* pointing to bus number )`

Pass in `MusicDeviceGroupID` for the AudioUnitElement.

Pass in `kAudioUnitScope_Group` for the AudioUnitScope.

`kMusicDeviceProperty_IsInInterrupt(UInt32* pointing to UInt32 where zero indicates "false" )`

This property is write-only -- we're telling the AudioUnit if it's in an interrupt

`kMusicDeviceProperty_Task(output data is ignored )`

```
kMusicDeviceProperty_QTMAInstrumentNumber
               (UInt32* pointing to inst number )
```

Parameters

```
enum
{
    kAudioUnitParameterUnit_Generic = 0, /* generic value generally
between 0.0 and 1.0 */
    kAudioUnitParameterUnit_Indexed = 1, /* takes an integer value (good
for menu selections) */
    kAudioUnitParameterUnit_Boolean = 2, /* 0.0 means FALSE, non-zero
means TRUE */
    kAudioUnitParameterUnit_Percent = 3, /* usually from 0 -> 100,
sometimes -50 -> +50 */
    kAudioUnitParameterUnit_Seconds = 4, /* absolute or relative time */
    kAudioUnitParameterUnit_SampleFrames = 5, /* one sample frame equals
(1.0/sampleRate) seconds */
    kAudioUnitParameterUnit_Phase = 6, /* -180 to 180 degrees */
    kAudioUnitParameterUnit_Rate = 7, /* rate multiplier, for playback
speed, etc. (e.g. 2.0 == twice as fast) */
    kAudioUnitParameterUnit_Hertz = 8, /* absolute frequency/pitch in
cycles/second */
    kAudioUnitParameterUnit_Cents = 9, /* unit of relative pitch */
    kAudioUnitParameterUnit_RelativeSemiTones = 10, /* useful for course
detuning */
    kAudioUnitParameterUnit_MIDINoteNumber = 11, /* absolute pitch as
defined in the MIDI spec (exact freq may depend on tuning table) */
    kAudioUnitParameterUnit_MIDIController = 12, /* a generic MIDI
controller value from 0 -> 127 */
    kAudioUnitParameterUnit_Decibels = 13, /* logarithmic relative gain
*/
    kAudioUnitParameterUnit_LinearGain  = 14, /* linear relative gain */
    kAudioUnitParameterUnit_Degrees = 15 /* -180 to 180 degrees, similar
to phase but more general (good for 3D coord system) */
};

typedef UInt32      AudioUnitParameterUnit;
```

Initialization

No substantial allocation of resources should occur when the AudioUnit component is opened. Instead, only access to basic properties is allowed. To fully initialize the AudioUnit, you call `AudioUnitInitialize()`.

AudioUnitInitialize

```
AudioUnitInitialize (AudioUnit  ci);
```

AudioUnitUninitialize

```
AudioUnitUninitialize (AudioUnit  ci);
```

Property Management

Properties describe or control some aspect of an AudioUnit that does not vary in time. Properties are addressed via their ID, scope, and element. The ID identifies the property and describes the structure of its data. The scope identifies the functional area of the AudioUnit of interest. The element identifies the specific part of the scope of interest.

Examples of properties include user readable names, stream format, data sources, and one shot configuration information.

AudioUnitGetPropertyInfo

```
AudioUnitGetPropertyInfo        (AudioUnit              ci,
                                 AudioUnitPropertyID    inID,
                                 AudioUnitScope         inScope,
                                 AudioUnitElement       inElement,
                                 UInt32 *               outDataSize,
                                 Boolean *              outWritable)
```

DISCUSSION

You can pass in NULL for outData, to determine how much memory to allocate for variable size properties.

AudioUnitGetProperty

```
AudioUnitGetProperty            (AudioUnit              ci,
                                 AudioUnitPropertyID    inID,
                                 AudioUnitScope         inScope,
                                 AudioUnitElement       inElement,
                                 void *                 outData,
                                 UInt32 *               ioDataSize);
```

AudioUnitSetProperty

```
AudioUnitSetProperty            (AudioUnit              ci,
                                 AudioUnitPropertyID    inID,
                                 AudioUnitScope         inScope,
                                 AudioUnitElement       inElement,
                                 void *                 inData,
                                 UInt32                 inDataSize);
```

AudioUnitSetRenderNotification

```
AudioUnitSetRenderNotification  (AudioUnit              ci,
                                 AudioUnitRenderCallback  inProc,
                                 void *                 inProcRefCon);
```

AudioUnitAddPropertyListener

```
AudioUnitAddPropertyListener    (AudioUnit              ci,
                                 AudioUnitPropertyID    inID,
                                 AudioUnitPropertyListenerProc  inProc,
                                 void *                 inProcRefCon);
```

AudioUnitRemovePropertyListener

```
AudioUnitRemovePropertyListener (AudioUnit              ci,
                                 AudioUnitPropertyID    inID,
                                 AudioUnitPropertyListenerProc  inProc);
```

Parameter Management

Parameters control a specific aspect of the processing of an AudioUnit that can vary in time. Parameters are represented as single floating point values and are addressed by their ID, scope and element similar to properties.

Since parameters vary in time, AudioUnits must support some notion of scheduling. That said, it is not the intent of this API to force an AudioUnit to support complete scheduling and history. It is assumed that the general workload of an AudioUnit's scheduler will revolve around scheduling events at most a few buffers into the future. Further, the event density is generally expected to be light. Therefore, the client of an AudioUnit should take care in the number of events it schedules as it could drastically affect performance.

Examples of parameters include volume, panning, filter cutoff, delay time LFO spead, and rate multiplier.

AudioUnitGetParameter

```
AudioUnitGetParameter           (AudioUnit              ci,
                                 AudioUnitParameterID   inID,
                                 AudioUnitScope         inScope,
                                 AudioUnitElement       inElement,
                                 Float32 *              outValue);
```

AudioUnitSetParameter

```
AudioUnitSetParameter           (AudioUnit              ci,
                                 AudioUnitParameterID   inID,
                                 AudioUnitScope         inScope,
                                 AudioUnitElement       inElement,
                                 Float32                inValue,
                                 UInt32
inBufferOffsetInFrames);
```

AudioUnitRenderSlice

```
AudioUnitRenderSlice (AudioUnit ci,
                      AudioUnitRenderActionFlags  inActionFlags,
                      const AudioTimeStamp * inTimeStamp,
                      UInt32 inOutputBusNumber,
                      AudioBuffer *  ioData);
```

If scope is global, then it reinitializes a device to its default state.

AudioUnitReset

```
AudioUnitReset                  (AudioUnit              ci,
                                 AudioUnitScope         inScope,
                                 AudioUnitElement       inElement);
```

Callbacks

The following is a callback that an AudioUnit can make from its RenderSlice function. It can be used as a RenderNotify callback for the owner (usually the AUGraph) to get called back before and after rendering.

It can also be used as an InputCallback to provide a buffer of audio inputto one of the AudioUnit's inputs.

The same arguments that are passed to `AudioUnitRenderSlice()` are passed on to the callback here.

```
typedef CALLBACK_API_C( OSStatus, AudioUnitRenderCallback )
                      (void *inRefCon, AudioUnitRenderActionFlags
                      inActionFlags, const AudioTimeStamp *inTimeStamp,
                      UInt32 inBusNumber, AudioBuffer *ioData);
```

The following callback is called when a client registers for notifications of property changes to an AudioUnit with the AudioUnitAddPropertyListener call.

```
typedef CALLBACK_API_C( void, AudioUnitPropertyListenerProc )
                        (void *inRefCon, AudioUnit ci,
                        AudioUnitPropertyID inID, AudioUnitScope inScope,
                        AudioUnitElement inElement);
```

Function Pointers

The following are function pointers defined for functions where performance is an issue.

You can use the `kAudioUnitProperty_FastDispatch` property to get a function pointer pointing directly to your implementation. This avoids the high cost of dispatching through the Component Manager.

```
typedef CALLBACK_API_C( ComponentResult, AudioUnitGetParameterProc )
                        (void *inComponentStorage, AudioUnitParameterID
                        inID, AudioUnitScope inScope, AudioUnitElement
                        inElement, Float32 *outValue);
```

```
typedef CALLBACK_API_C( ComponentResult, AudioUnitSetParameterProc )
                        (void *inComponentStorage, AudioUnitParameterID
                        inID, AudioUnitScope inScope, AudioUnitElement
                        inElement, Float32 inValue, UInt32
                        inBufferOffsetInFrames);

typedef CALLBACK_API_C( ComponentResult , AudioUnitRenderSliceProc )
                        (void *inComponentStorage,
                        AudioUnitRenderActionFlags inActionFlags, const
                        AudioTimeStamp *inTimeStamp, UInt32
                        inOutputBusNumber, AudioBuffer *ioData);
```

Errors

```
enum {
    kAudioUnitErr_InstrumentTypeNotFound = -10872,
    kAudioUnitErr_InvalidFile = -10871,
    kAudioUnitErr_UnknownFileType = -10870,
    kAudioUnitErr_FileNotSpecified = -10869,
    kAudioUnitErr_InvalidProperty = -10879,
    kAudioUnitErr_InvalidParameter = -10878,
    kAudioUnitErr_InvalidElement = -10877,
    kAudioUnitErr_NoConnection= -10876,
    kAudioUnitErr_FailedInitialization = -10875,
    kAudioUnitErr_TooManyFramesToProcess = -10874,
    kAudioUnitErr_IllegalInstrument = -10873,
    kAudioUnitErr_FormatNotSupported = -10868,
    kAudioUnitErr_Uninitialized= -10867
};
```

CHAPTER 4

Audio Toolbox

This chapter discusses the AUGraph and Music Sequence APIs, which are part of the Audio Toolbox on Mac OS X, and the services provided by the `AudioToolbox.framework` that applications can use for audio processing. The section "Reference" (page 53) describes the constants, data types and functions that comprise the Audio Toolbox framework

Overview

The `AudioToolbox.framework` provides a set of services that applications can use for audio processing. The framework is divided into the following groups of APIs:

- `AUGraph.h`
- `MusicPlayer.h`

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

The AUGraph

The AUGraph is a high-level representation of a set of AudioUnits, along with the connections between them. You can use these APIs to construct arbitrary signal paths through which audio may be processed, i.e., a modular routing system. The APIs deal with large numbers of AudioUnits and their relationships.

AudioGraphs provide the following services:

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

AUGraphs are created and destroyed using the `NewAUGraph()` and `DisposeAUGraph()` calls.

AUGraph APIs

`AUGraph` is an object that provides services to create graphs of audio units, i.e., a signal processing graph. An `AUGraph` must have some kind of Audio Output unit as its head, so that when `AUGraphStart` is called, the `AUGraph` calls the start method of the output unit to start the work of the `AUGraph`. This output unit is then responsible for driving the production of audio data by its graph.

For instance, in the case of an audio output unit that talks to an `AudioDevice`, this will start (and stop) the `IOProc` cycle of that device, and the production audio data is constrained by the time period of the device, i.e., the processing must be done in real time.

Other output units could be written that would write their data to a file. In that case, the start and stop methods would be responsible for preparing the file to be written and establish the cycle of calls that would pull on the `AUGraph`'s `AudioUnit` nodes. In this case, the processing of the graph is not constrained by time, but perhaps by the amount of disk space available to write the file to.

`AUGraph` objects can also be serialized, with their state written to a data file, and then re-established from that data. The format of that data stream is not public at this time.

AUGraph State

The AUGraph maintains its representation using the AUNode structure, even when the AudioUnit components themselves are not instantiated.

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

The AUGraph APIs are responsible for representing the description of a set of AudioUnit 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 AudioUnits (AUGraphOpen()), and making the physical connections between them stored in the representation (AUGraphInitialize()). Thus, the AUGraph is a description of the various AudioUnits and their connections, but also manage the actual instantiated AudioUnits.

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 AudioUnits in the graph. The various nodes (AUNode) in the AUGraph representing AudioUnits may be added or removed, and the connections between them modified.

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

This class data is in an arbitrary format, and may differ depending on the particular `AudioUnit`. In general, the data is used by the `AudioUnit` 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.

The MusicPlayer API

The `MusicPlayer.h` APIs provide the services of a sequencing toolbox. This toolbox is where events can be collected into tracks, and tracks can be copied, pasted, and looped within a sequence.

The Music Player API is described in detail in Chapter 9, "Audio Toolbox Reference."

Each `MusicSequence` object can be associated with a single `AUGraph` object, and it is the nodes within that graph that the sequence will generally interact with, directing its events to particular nodes. A `MusicSequence` is played by a `MusicPlayer` object.

A `MusicSequence` can be created from a Standard MIDI File (`.smf` or `.mid`). When using this, the sequence will create tracks that correspond to the different MIDI channels that have events associated with them in the MIDI file.

The `MusicSequence` track's can be iterated over, allowing your application to scan through the events that are contained with that track.

There are a set of predefined event types in this API — including a user event type — that applications can use for their own custom events.

A `MusicSequence` contains an arbitrary number of tracks (`MusicTrack`) each of which contains time-stamped (typically in units of beats, or seconds) events in

time-increasing order. There are various types of events, including the expected MIDI events, tempo, and extended events.

A `MusicTrack` has properties which may be inspected and assigned, including support for looping, muting/soloing, and timestamp interpretation. There are APIs for iterating through the events in a `MusicTrack` and for performing basic editing operations on them.

Each `MusicSequence` may have an associated AUGraph object, which represents a set of AudioUnits and the connections between them. Thus, each MusicTrack of the MusicSequence may address its events to a specific `AudioUnit` within the `AUGraph`.

Consequently, you can automate arbitrary parameter changes to AudioUnits, and schedule notes to be played to MusicDevices (AudioUnit software synthesizers) within an arbitrary audio processing network (AUGraph).

`MusicSequence` global information consists of:

- An AUGraph.
- Copyright and other textual information.

`MusicTrack` properties are:

- AUNode (in the AUGraph) of the AudioUnit addressed by the MusicTrack.
- Textual information.
- Mute / solo state.
- Offset time.
- Loop time and number of loops.
- Time units for the event timestamps (beats, seconds, ...).
- Beats go through tempo map, seconds map absolute time.

Reference

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

Types

AUGraph

```
typedef long    AUNode;
typedef struct OpaqueAUGraph    *AUGraph;
```

MusicSequence

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

struct MIDIChannelMessage
{
    UInt8       status;     // contains message and channel

    // message specific data
    UInt8       data1;
    UInt8       data2;
    UInt8       reserved;
};
```

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

struct MIDIMetaEvent
{
    UInt8       metaEventType;
    UInt32      dataLength;
    UInt8       data[1];
};

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

// allocated space for 16 arguments
struct ExtendedNoteOnEvent16
{
    MusicDeviceInstrumentID     instrumentID;
    MusicDeviceGroupID          groupID;
    Float32                     duration;
    MusicDeviceNoteParams16     extendedParams;
};

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

struct ExtendedTempoEvent
{
    Float64     bpm;
};
```

```
typedef struct OpaqueMusicPlayer        *MusicPlayer;
typedef struct OpaqueMusicSequence      *MusicSequence;
typedef struct OpaqueMusicTrack         *MusicTrack;
typedef struct OpaqueMusicEventIterator *MusicEventIterator;
```

Constants:AUGraph

```
enum {
    kAUGraphErr_NodeNotFound = -10860
};
```

MusicSequence

```
enum
{
    kSequenceTrackProperty_LoopInfo = 0,    // struct {MusicTimeStamp
loopLength; long numberOfLoops;};
    kSequenceTrackProperty_OffsetTime = 1,  // struct {MusicTimeStamp
offsetTime;};
    kSequenceTrackProperty_MuteStatus = 2,  // struct {Boolean
muteState;};
    kSequenceTrackProperty_SoloStatus = 3   // struct {Boolean
soloState;};
};
```

Depending on the event type, you cast the returned `void*` pointer to:

```
kMusicEventType_ExtendedNoteExtendedNoteOnEvent*

        kMusicEventType_ExtendedControl         ExtendedControlEvent*
        kMusicEventType_ExtendedTempo           ExtendedTempoEvent*
        kMusicEventType_User                    <user-defined-data>*
        kMusicEventType_Meta                    MIDIMetaEvent*
```

```
    kMusicEventType_MIDINoteMessage         MIDINoteMessage*
    kMusicEventType_MIDIChannelMessage      MIDIChannelMessage*
    kMusicEventType_MIDIRawData             MIDIRawData*
```

Defining the Events Supported by the Sequencer

The following music event types, including both MIDI and "extended" protocol, are supported by the sequencer:

```
enum
{
    kMusicEventType_NULL                = 0,
    kMusicEventType_ExtendedNote,       // note with variable number of
                                            arguments (non-MIDI)
    kMusicEventType_ExtendedControl,    // control change (non-MIDI)
    kMusicEventType_ExtendedTempo,      // tempo change in BPM
    kMusicEventType_User,               // user defined data
    kMusicEventType_Meta,               // standard MIDI file meta event
    kMusicEventType_MIDINoteMessage,    // MIDI note-on with duration
                                            (for note-off)
    kMusicEventType_MIDIChannelMessage, // MIDI channel messages (other
                                            than note-on/off)
    kMusicEventType_MIDIRawData,        // for system exclusive data
    kMusicEventType_Last                // always keep at end
};

typedef UInt32      MusicEventType;
typedef Float64     MusicTimeStamp;

#define kMusicTimeStamp_EndOfTrack1000000000.0
```

You pass this value in to indicate a time passed the last event in the track. In this way, it is possible to perform edits on tracks which include all events starting at some particular time up to and including the last event.

```
enum
{
    kAudioToolboxErr_TrackIndexError        = -10859,
    kAudioToolboxErr_TrackNotFound          = -10858,
```

```
    kAudioToolboxErr_EndOfTrack             = -10857,
    kAudioToolboxErr_StartOfTrack           = -10856
};
```

Functions

NewAUGraph

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

DisposeAUGraph

```
DisposeAUGraph( AUGraphinGraph );
```

AUGraphNewNode

```
AUGraphNewNode(AUGraphinGraph,
               ComponentDescription *inDescription, UInt32
               inClassDataLength, const void*inClassData,
               AUNode *outNode);
```

DISCUSSION

If the node has no associated class data, pass in zero for `inClassDataLength`, and NULL for `inClassData`.

AUGraphRemoveNode

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

AUGraphGetNodeCount

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

AUGraphGetIndNode

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

AUGraphGetNodeInfo

```
AUGraphGetNodeInfo( AUGraphinGraph, AUNodeinNode,
                    ComponentDescription*outDescription,// pass in NULL
                    if not interested
                    UInt32*outClassDataLength,// pass in NULL if not
                    interested
                    void**outClassData,// pass in NULL if not interested
                    AudioUnit *outAudioUnit /* 0 if component not loaded
                    (graph is not wired) */ );
```

AUGraphConnectNodeInput

Connects a node's output to a node's input.

```
AUGraphConnectNodeInput(AUGraphinGraph,
                    AUNodeinSourceNode,
                    UInt32inSourceOutputNumber,
                    AUNodeinDestNode,
                    UInt32inDestInputNumber);
```

AUGraphDisonnectNodeInput

Disconnects a node's input.

```
AUGraphDisonnectNodeInput(AUGraphinGraph,
                    AUNodeinDestNode,
                    UInt32inDestInputNumber);
```

AUGraphClearConnections

Clears all connections of all nodes.

```
AUGraphClearConnections( AUGraphinGraph );
```

AUGraphUpdate

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

DISCUSSION

You call this function after performing a series of "edits" on the AUGraph with calls such as `AUGraphConnectNodeInput()` to finalize.

The call will be synchronous if `outIsUpdated` is NULL, meaning that it will block until the changes are incorporated into the graph if `outIsUpdated` is non-NULL. Then `AUGraphUpdate()` will return immediately and `outIsUpdated` will equal `true` if the changes were already made (no more changes to make) or `false` if the changes are still outstanding.

The following calls must be made in this order:

1. You instantiate the graph from the representation (opens all AudioUnits) by calling the `AUGraphOpen` function.
2. You fully initialize the AudioUnits (you call `AudioUnitInitialize()` on each) to prepare for audio processing by calling the `AUGraphInitialize` function.
3. You instruct the graph to start rendering by calling the `AUGraphStart` function.
4. You instruct the graph to stop rendering by calling the `AUGraphStop` function.
5. You uninitialize the AudioUnits (you call `AudioUnitUninitialize()` on each) without closing the components by calling the `AUGraphUnInitialize` function.
6. You destroy the built graph, leaving only the representation (this closes all AudioUnits) by calling the `AUGraphClose` function.

Graphs can be started/stopped, inited/uninited, opened/closed, based on usage requirements of the user.

The following are query APIs:

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

AUGraphIsInitialized(AUGraph inGraph,
                     Boolean *outIsInitialized );

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

Music Player Transport APIs

NewMusicPlayer

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

DisposeMusicPlayer

```
DisposeMusicPlayer( MusicPlayerinPlayer );
```

MusicPlayerSetSequence

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

MusicPlayerSetTime

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

MusicPlayerGetTime

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

MusicPlayerPreroll

Allows synth devices to load instrument samples.

```
MusicPlayerPreroll( MusicPlayer inPlayer);
```

MusicPlayerStart

```
MusicPlayerStart(   MusicPlayer inPlayer);
```

MusicPlayerStop

```
MusicPlayerStop(    MusicPlayer inPlayer);
```

Music Sequence APIs

NewMusicSequence

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

DisposeMusicSequence

```
DisposeMusicSequence(MusicSequenceinSequence );
```

MusicSequenceNewTrack

Create a new track in the sequence.

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

MusicSequenceDisposeTrack

Removes the track from a sequence and disposes the track.

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

MusicSequenceGetTrackCount

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

MusicSequenceGetIndTrack

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

MusicSequenceGetTrackIndex

Returns error code if track is not found in the sequence.

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

MusicSequenceSetAUGraph

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

MusicSequenceGetAUGraph

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

MusicSequenceLoadSMF

```
MusicSequenceLoadSMF(MusicSequence inSequence,
                    FSSpec *inFileSpec );
```

DISCUSSION

Standard MIDI files (SMF, and RMF). This function also intelligently parses an RMID file to extract SMF part inResolution is relationship between "tick" and quarter note for saving to SMF.

MusicSequenceSaveSMF

```
MusicSequenceSaveSMF(MusicSequence inSequence,
                     FSSpec *inFileSpec,
                     UInt16inResolution );
```

DISCUSSION

Pass in zero to use default (480 PPQ, normally).

MusicSequenceReverse

Reverses (in time) all events (including tempo events).

```
MusicSequenceReverse(MusicSequence inSequence);
```

MusicTrack APIs

MusicTrackGetSequence

```
MusicTrackGetSequence(MusicTrack inTrack,
                     MusicSequence*outSequence );
```

MusicTrackSetDestNode

```
MusicTrackSetDestNode(MusicTrack inTrack,
                     AUNodeinNode );
```

SequenceTrack Property APIs

MusicTrackSetProperty

```
MusicTrackSetProperty(MusicTrack inTrack,
                      UInt32 inPropertyID,
                      void*inData,
                      UInt32inLength );
```

DISCUSSION

The `inLength` parameter is currently ignored for the properties with fixed size.

MusicTrackGetProperty

```
MusicTrackGetProperty(MusicTrack inTrack,
                      UInt32 inPropertyID,
                      void*inData,
                      UInt32*ioLength );
```

DISCUSSION

If `inData` is NULL, then the length of the data will be passed back in outLength. This allows the client to allocate a buffer of the correct size (useful for variable length properties -- currently all properties have fixed size).

Notes on properties:

`kSequenceTrackProperty_LoopInfo`

The default looping behavior is to loop once through the entire track pass zero in for inNumberOfLoops to loop forever.

Editing Operations on Sequence Tracks

All time ranges are as follows [starttime, endtime). The range includes the start time, but includes events only up to, but *not* including the end time.

MusicTrackMoveEvents

```
MusicTrackMoveEvents(MusicTrack inTrack,
                     MusicTimeStampinStartTime,
                     MusicTimeStampinEndTime,
                     MusicTimeStampinMoveTime );
```

DISCUSSION

`inMoveTime` may be negative to move events backwards in time.

NewMusicTrackFrom

```
NewMusicTrackFrom(  MusicTrackinSourceTrack,
                     MusicTimeStampinSourceStartTime,
                     MusicTimeStampinSourceEndTime,
                     MusicTrack *outNewTrack );
```

MusicTrackClear

Removes all events in the given range.

```
MusicTrackClear(    MusicTrack inTrack,
                     MusicTimeStampinStartTime,
                     MusicTimeStampinEndTime );
```

MusicTrackCut

```
MusicTrackCut(      MusicTrack inTrack,
                    MusicTimeStampinStartTime,
                    MusicTimeStampinEndTime );
```

DISCUSSION

This is the same as `MusicTrackClear()`, but also moves all following events back by the range's duration.

MusicTrackCopyInsert

```
MusicTrackCopyInsert(MusicTrack inSourceTrack,
                     MusicTimeStampinSourceStartTime,
                     MusicTimeStampinSourceEndTime,
                     MusicTrack inDestTrack,
                     MusicTimeStampinDestInsertTime );
```

DISCUSSION

The given source range is inserted at inDestInsertTime in inDestTrack (all events at and after inDestInsertTime in inDestTrack are moved forward by the range's duration).

MusicTrackMerge

```
MusicTrackMerge(    MusicTrack inSourceTrack,
                    MusicTimeStampinSourceStartTime,
                    MusicTimeStampinSourceEndTime,
                    MusicTrack inDestTrack,
                    MusicTimeStampinDestInsertTime );
```

DISCUSSION

The given source range is merged with events starting at `inDestInsertTime` in `inDestTrack`.

Sequence Track Event Access and Manipulation

The following routines can be used for sequence track event access and manipulation.

NewMusicEventIterator

```
NewMusicEventIterator(MusicTrack inTrack,
                      MusicEventIterator*outIterator );
```

DISCUSSION

Event iterator objects on tracks.

DisposeMusicEventIterator

```
DisposeMusicEventIterator(MusicEventIteratorinIterator );
```

MusicEventIteratorSeek

```
MusicEventIteratorSeek(MusicEventIterator inIterator,
                      MusicTimeStamp inTimeStamp );
```

DISCUSSION

Passing in `kMusicTimeStamp_EndOfTrack` for inBeat will position "iterator" to the end of track (which is pointing to the space just AFTER the last event). You can

use the `MusicEventIteratorPreviousEvent` function to backup one, if you want last event.

MusicEventIteratorNextEvent

Seeks track "iterator" to the next event.

```
MusicEventIteratorNextEvent(MusicEventIterator inIterator);
```

MusicEventIteratorPreviousEvent

Seeks track iterator to the previous event (if the iterator is already at the first event, then it remains unchanged and an error code is returned).

```
MusicEventIteratorPreviousEvent(MusicEventIterator inIterator);
```

MusicEventIteratorGetEventInfo

Returns an error code if the track's "iterator" is currently at the end of the track.

```
MusicEventIteratorGetEventInfo(MusicEventIterator inIterator,
                    MusicTimeStamp*outTimeStamp,
                    MusicEventType*outEventType,
                    void**outEventData,
                    UInt32*outEventDataSize );
```

Deleting Events

The following APIs can be used to delete the event at the current iterator.

MusicEventIteratorDeleteEvent

```
MusicEventIteratorDeleteEvent(MusicEventIterator inIterator);
```

MusicEventIteratorSetEventTime

```
MusicEventIteratorSetEventTime(MusicEventIterator inIterator,
                    MusicTimeStampinTimeStamp );
```

MusicEventIteratorHasPreviousEvent

```
MusicEventIteratorHasPreviousEvent(MusicEventIterator inIterator,
                    Boolean*outHasPreviousEvent );
```

MusicEventIteratorHasNextEvent

```
MusicEventIteratorHasNextEvent(MusicEventIterator inIterator,
                    Boolean*outHasNextEvent );
```

Adding Time-Stamped Events

The following APIs are used to add time-stamped events to the track.

MusicTrackNewMIDINoteEvent

```
MusicTrackNewMIDINoteEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    const MIDINoteMessage*inMessage );
```

MusicTrackNewMIDIChannelEvent

```
MusicTrackNewMIDIChannelEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    const MIDIChannelMessage *inMessage );
```

MusicTrackNewMIDIRawDataEvent

```
MusicTrackNewMIDIRawDataEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    const MIDIRawData*inRawData );
```

MusicTrackNewExtendedNoteEvent

```
MusicTrackNewExtendedNoteEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    const ExtendedNoteOnEvent*inInfo );
```

MusicTrackNewExtendedControlEvent

```
MusicTrackNewExtendedControlEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    const ExtendedControlEvent*inInfo );
```

MusicTrackNewExtendedTempoEvent

```
MusicTrackNewExtendedTempoEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    Float64inBPM);
```

MusicTrackNewMetaEvent

```
MusicTrackNewMetaEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    void*inMetaEventInfo,
                    UInt32inMetaEventLength );
```

MusicTrackNewUserEvent

```
MusicTrackNewUserEvent(MusicTrack inTrack,
                    MusicTimeStampinTimeStamp,
                    void*inUserData,
                    UInt32inUserDataLength );
```

Event Representation and Manipulation Within a Track

You need to be careful in dealing with both SMF-types of MIDI events, and also be upwardly compatible with an extended MPEG4-SA like paradigm. The solution is to hide the internal event representation from the client and allow access to the events through accessor functions. In so doing, the user can examine and create standard events, or any user-defined event.

CHAPTER 5

MIDI System Services

This chapter discusses the MIDI System Services available on Mac OS X. The section "Reference" (page 80) describes the constants, data types and functions that comprise the CoreMIDI framework.

Overview

In Mac OS X, Apple provides a new set of system services, so that applications and MIDI hardware can communicate in a single unified way, using a single API.

MIDI services, which are low level, provide high-performance access to MIDI hardware devices. There is a driver model in the MIDI world that "talks" directly to IOKit, so your application has a direct path from the MIDI services API to the hardware.

Using this driver model, third-party manufacturers can create driver plugins that talk to IOKit. Those can then be loaded and managed by a server, which applications talk to through the Core MIDI framework.

The CoreMIDI framework provides the client-side API that applications use to interface to MIDI devices. The CoreMIDIServer framework is used by those developers who provide drivers for MIDI devices, and is not covered explicitly in this document.

In Java, these services are available in the `com.apple.audio.midi` package.

Goals

The primary goal of MIDI services in Mac OS X is to have interoperability between applications and hardware, so that everyone is working to the same

standard. Other goals include providing MIDI I/O with highly accurate timing, as required by professional applications. This means from a musical point of view being able to get a MIDI event into and out of the computer within one millisecond, i.e., to keep latency under one millisecond, and also to keep jitter — i.e., the variations in I/O — under 200 microseconds.

Another goal is to provide a single system-wide configuration, i.e., knowing what devices are present, and being able to assign names to those devices, manufacturer names, and what MIDI channels they're receiving on and so on.

The MIDI services are designed as an extensible system. Toward that end, a device can have any number of properties attached to it. And a device manufacturer can publish their particular properties of their device.

Implementation

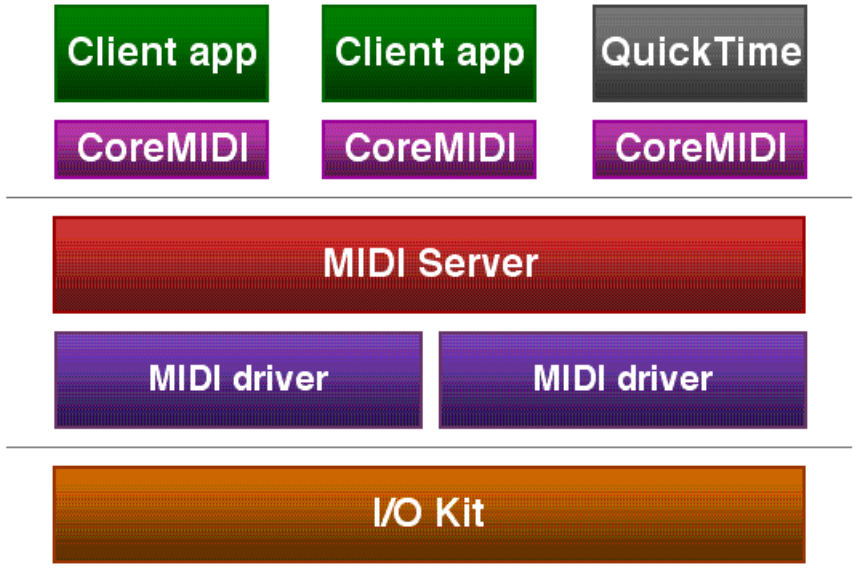
The client API is implemented as the CoreMIDI framework, which is built as a Mach-O library. Figure 5-1 shows the MIDI implementation.

The CoreMIDI framework is what applications link to. That framework is responsible for loading the MIDI Server, if it isn't already running, and communicating with Mach messages to get messages across the process boundaries. A driver will link with CoreMIDIServer, which provides the same API, except that the driver is running inside the server's address space. Instead of having to go through Mach-O messages to talk to the service, it has direct access to the API.

The server process loads, and manages all communication with MIDI drivers. Most of its implementation is in the CoreMIDIServer framework, which drivers may import in order to access the API. Drivers are not I/O Kit drivers, but rather, are dynamic libraries, using CFPlugin.

Many MIDI drivers can simply be user-side I/O Kit clients (probably for serial, USB, Firewire). PCI card drivers will need their MIDI drivers to communicate with a separate Kernel extension.

Figure 5-1 An architectural diagram of the MIDI implementation on Mac OS X



MIDI Drivers

Apple provides in Mac OS X a USB MIDI Class standard drivers.

Drivers are installed in `/System/Library/Extensions`. They are loaded and managed by the MIDI server, and implemented as user-space CFPlugIns. No kernel extension (“real” driver) is necessary, unless required by the I/O needs of the device — for example, a PCI card.

The MIDI driver functions have a simple programming interface and detect the presence of hardware. You create a MIDIDevice, MIDIEntity and MIDIEndpoint object and set their properties. You can perform MIDI I/O, using I/O Kit.

MIDI Hardware

Some examples of MIDI hardware include

- USB MIDI interfaces.
- USB synthesizers and keyboards. For example, Roland has introduced a synthesizer that connects directly to the computer by means of USB.

- PCI cards that have MIDI connectors.
- FireWire MIDI interfaces.

CoreMIDI Objects

MIDI I/O is based on the concept of a source or destination `MIDIEndPoint`, where this endpoint is a complete MIDI stream, i.e., 16 channels of MIDI data, System messages, and so on, as defined by the MIDI specification.

A `MIDIDevice` represents a piece of MIDI hardware. MIDI I/O devices have become increasingly complex primarily in order to solve the problem that 16 channels of data is not enough to deal with the multi-timbral requirements of music composition and studio usage. Thus, many devices present multiple MIDI sources and destinations that are presented to the computer as intertwined data streams. `MIDIEntity`s' represent a logical grouping of functionality as defined by the MIDI driver for that device.

For instance, a device with a MIDI-In and MIDI-Out plug and a GM synthesizer could be presented as a device comprised of two `MIDIEntity` objects — one for the I/O plugs, the other for the built-in synthesizer. The device would be presented to the application as three `MIDIEndpoint`s, two inputs (MIDI-Out plug and GM Synthesizer), and one output (MIDI-In plug) for the application.

To read and write MIDI data, an application uses the services of `MIDIPort` objects. These objects facilitate the moving of MIDI data within the computer and in and out. Typically, an application creates a single MIDI in or out port, and then uses this to set sources or destinations to particular `MIDIEndPoint`s.

`MIDIEndPoints` can also be virtual. In that case, they represent another application on the system that is either sending or receiving MIDI data, thus enabling applications to route MIDI data between each other — i.e., the Inter Application Communication of MIDI data.

MIDIPacketList

The MIDIPacketList are a time-stamped list of packets to or from one endpoint. All MIDI I/O functions use this structure. The MIDIPacket contains one or more simultaneous MIDI events. The exception is a system-exclusive events.

Iterating Through a MIDIPacketList

The following code snippet illustrates how to build a MIDIPacketList, and add and event to the list.

```
Byte buffer[1024];
MIDIPacketList *pktlist = (MIDIPacketList *)buffer;

MIDIPacket *curPacket = MIDIPacketListInit(pktlist);
Byte noteOn[] = { 0x90, 60, 64 };

curPacket = MIDIPacketListAdd(pktlist, sizeof(buffer), curPacket,
        timeStamp, 3, noteOn);
```

MIDIPacketList helper functions:

```
// iterate through the packets in a packet list
MIDIPacket *MIDIPacketNext(MIDIPacket *pkt);

// begin building a packet list
MIDIPacket *MIDIPacketListInit(MIDIPacketList *pktlist);

// add an event to a packet list
MIDIPacket *MIDIPacketListAdd(MIDIPacketList *pktlist, ByteCount
listSize, MIDIPacket *curPacket, MIDITimeStamp time, ByteCount nData,
Byte *data);
```

To find and send to destinations:

```
ItemCount nDests = MIDIGetNumberOfDestinations();
for (int iDest = 0; iDest < nDests; ++iDest) {

    MIDIEndpointRef dest = MIDIGetDestination(iDest);
    MIDISend(gOutputPort, dest, &packetList);
}
```

To find and receive from sources:

```
ItemCount nSrcs = MIDIGetNumberOfSources();
for (int iSrc = 0; iSrc < nSrcs; ++iSrc) {
    MIDIEndpointRef src = MIDIGetSource(iSrc);
    void *srcConnRefCon = src;
```

```
    MIDIPortConnectSource(inPort, src,
    srcConnRefCon );
}

void MyReadProc(const MIDIPacketList *pktlist,
void *readProcRefCon, void *srcConnRefCon);
```

Using MIDIReadProc

You use a callback function to receive MIDI input. This is called in a high-priority thread created by CoreMIDI.framework.

Iterating through devices and entities:

```
    ItemCount ndev = MIDIGetNumberOfDevices();
    for (int idev = 0; idev < ndev; ++idev) {
        MIDIDeviceRef device = MIDIGetDevice(idev);
        ItemCount nent =
            MIDIGetNumberOfEntities(device);
        for (int ient = 0; ient < nent; ++ient) {
    MIDIEntityRef entity =
        MIDIGetEntity(device, ient);
...
    }
    }
```

Reference

This reference section describes the constants, data types and functions that comprise the CoreMIDI framework available on Mac OS X. The functions are topically arranged according to usage and correspond to the ordering in the `CoreMIDI.h` file.

Types

Opaque Types

```
typedef MIDIObjectRef
```

MIDIObject is the base class for many of the objects in CoreMIDI. They have properties, and often an "owner" object, from which they inherit any properties they do not themselves have.

Developers may add their own private properties, whose names must begin with their company's inverted domain name, as in Java package names, but with underscores instead of dots, for example:
`com_apple_APrivateAppleProperty`

```
typedef void *MIDIObjectRef;
```

```
typedef MIDIClientRef
```

Derives from `MIDIObjectRef`; does not have an owner object.

To use CoreMIDI, an application creates a MIDIClientRef, to which it can add MIDIPortRefs, through which it can send and receive MIDI.

```
typedef struct OpaqueMIDIClient *MIDIClientRef;
```

```
typedef MIDIPortRef
```

Derives from MIDIObjectRef; owned by a `MIDIClientRef`.

A `MIDIPortRef`, which may be an input port or output port, is an object through which a client may communicate with any number of MIDI sources or destinations.

```
typedef struct OpaqueMIDIPort * MIDIPortRef;
```

```
typedef  MIDIDeviceRef
```

Derives from `MIDIObjectRef`; does not have an owner object.

A MIDI device, which either attaches directly to the computer and is controlled by a MIDI driver, or which is "external," meaning that it is connected to a driver-controlled device via a standard MIDI cable.

A `MIDIDeviceRef` has properties and contains MIDIEntityRefs.

```
typedef struct OpaqueMIDIDevice *MIDIDeviceRef;

typedef MIDIEntityRef
```

Derives from `MIDIObjectRef`, owned by a `MIDIDeviceRef`.

Devices may have multiple logically distinct sub-components, for example, a MIDI synthesizer and a pair of MIDI ports, both addressable via a USB port.

By grouping a device's endpoints into entities, the system has enough information for an application to make reasonable assumptions about how to communicate in a bi-directional manner with each entity, as is desirable in MIDI librarian applications.

These sub-components are MIDIEntityRef's.

```
typedef struct OpaqueMIDIEntity *MIDIEntityRef;

typedef MIDIEndpointRef
```

Derives from `MIDIObjectRef`, owned by a `MIDIEntityRef`.

Entities have any number of MIDIEndpointRef's, sources and destinations of 16-channel MIDI streams.

```
typedef struct OpaqueMIDIEndpoint *MIDIEndpointRef;
```

Forward Structure Declarations

```
typedef struct MIDIPacketList           MIDIPacketList;
typedef struct MIDISysexSendRequest     MIDISysexSendRequest;
typedef struct MIDINotification         MIDINotification;

typedef MIDITimeStamp
```

A host clock time representing the time of an event, as returned by `AudioGetCurrentHostTime()` (or `UpTime()`).

Since MIDI applications will tend to do a fair amount of math with the times of events, it's more convenient to use a UInt64 than an AbsoluteTime.

```
typedef UInt64MIDITimeStamp;
```

Callback Functions

```
typedef MIDINotifyProc
```

This callback function is called when some aspect of the current MIDI setup changes. Currently, the only defined message is kMIDIMsgSetupChanged, which simply means, something changed. msgData is null in this case.

`message` A structure containing information about what changed.

`refCon` The client's refCon passed to MIDIClientCreate.

```
typedef void

(*MIDINotifyProc)(const MIDINotification *message, void *refCon);
```

```
typedef MIDIReadProc
```

This is a callback function through which a client receives incoming MIDI messages.

A MIDIReadProc function pointer is passed to the MIDIInputPortCreate and MIDIDestinationCreate functions. The CoreMIDI framework will create a high-priority receive thread on your client's behalf, and from that thread, your MIDIReadProc will be called when incoming MIDI messages arrive. Because this function is called from a separate thread, be aware of the synchronization issues when accessing data in this callback.

`pktlist` The incoming MIDI message(s).

`readProcRefCon`

The refCon you passed to MIDIInputPortCreate or MIDIDestinationCreate.

`srcConnRefCon`

A refCon you passed to MIDIPortConnectSource, which identifies the source of the data.

```
typedef void

(*MIDIReadProc)(const MIDIPacketList *pktlist, void *readProcRefCon, void
*srcConnRefCon);
```

```
typedef MIDICompletionProc
```

A callback function to notify the client of the completion of a call to MIDISendSysex.

`request` The MIDISysexSendRequest which has completed, or been aborted.

```
typedef void

(*MIDICompletionProc)(MIDISysexSendRequest *request);
```

Structs

```
struct      MIDIPacket
```

One or more MIDI events occuring at a particular time.

Field descriptions

`timeStamp` The time at which the events occurred, if receiving MIDI, or, if sending MIDI, the time at which the events are to be played. Zero means now.

`length` The number of valid MIDI bytes which follow, in data. (It may be larger than 256 bytes if the packet is dynamically allocated.)

`data` A variable-length stream of MIDI messages. Running status is not allowed. In the case of system-exclusive messages, a packet may only contain a single message, or portion of one, with no other MIDI events.

(This is declared to be 256 bytes in length so clients don't have to create custom data structures in simple situations.)

```
struct MIDIPacket
{
    MIDITimeStamp       timeStamp;
    UInt16              length;
    Byte                data[256];
};
typedef struct MIDIPacket           MIDIPacket;

struct      MIDIPacketList
```

A list of MIDI events being received from, or being sent to, one endpoint. Note that the packets, while defined as an array, may not be accessed as an array, since they are variable-length. To iterate through the packets in a packet list, use a loop such as:

```
<pre>
MIDIPacket *packet = &packetList->packet[0];
for (int i = 0; i < packetList->numPackets; ++i) {
...
packet = MIDIPacketNext(packet);
}
</pre>
```

Field descriptions

`numPackets`	The number of MIDIPackets in the list.
`packet`	An open-ended array of variable-length MIDIPackets.

```
struct MIDIPacketList
{
    UInt32              numPackets;
    MIDIPacket          packet[1];
};
typedef struct MIDIPacketList       MIDIPacketList;

struct      MIDISysexSendRequest
```

An asynchronous request to send a single system-exclusive MIDI event to a MIDI destination.

Field descriptions

`destination`	The endpoint to which the event is to be sent.
`data`	Initially, a pointer to the sys-ex event to be sent. MIDISendSysex will advance this pointer as bytes are sent.
`bytesToSend`	Initially, the number of bytes to be sent. MIDISendSysex will decrement this counter as bytes are sent.
`complete`	The client may set this to true at any time to abort transmission. The implementation sets this to true when all bytes have been sent.
`completionProc`	Called when all bytes have been sent, or after the client has set complete to true.
`completionRefCon`	Passed as a refCon to completionProc.

```
struct MIDISysexSendRequest

{
    MIDIEndpointRef     destination;
    Byte *              data;
    UInt32              bytesToSend;
    Boolean             complete;
    Byte                reserved[3];
    MIDICompletionProc  completionProc;
    void *              completionRefCon;
};

typedef struct MIDISysexSendRequestMIDISysexSendRequest;

struct      MIDINotification
```

A MIDINotification is a structure passed to a MIDINotifyProc, when CoreMIDI wishes to inform a client of a change in the state of the system.

Field descriptions

`messageID`	Type of message.
`messageSize`	Size of the entire message, including messageID and messageSize.

```
    SInt32                  messageID;
    ByteCount                   messageSize;
    // Additional data may follow, depending on messageID
};

struct MIDINotification
{
typedef struct MIDINotificationMIDINotification;

enum        MIDINotificationMessageID's

kMIDIMsgSetupChanged
```

Some aspect of the current MIDISetup has changed. `msgData` is NULL.

```
enum {
    kMIDIMsgSetupChanged = 1            // msgData is NULL
};
```

Property Name Constants

`kMIDIPropertyName`

Device/entity/endpoint property, string.

```
CFStringRefkMIDIPropertyName;
```

`kMIDIPropertyManufacturer`

Device/entity/endpoint property, string.

```
CFStringRefkMIDIPropertyManufacturer;
```

`kMIDIPropertyModel`

Device/entity/endpoint property, string.

```
CFStringRefkMIDIPropertyModel;
```

`kMIDIPropertyUniqueID`

Devices, entities, endpoints all have unique ID's, integer.

```
CFStringRefkMIDIPropertyUniqueID;
```

```
kMIDIPropertyDeviceID
```

Entity/endpoint property, integer.

The entity's system-exclusive ID, in user-visible form.

```
CFStringRefkMIDIPropertyDeviceID;
```

```
kMIDIPropertyReceiveChannels
```

Endpoint property, integer.

Set by the owning driver; should not be touched by other clients. The value is a bitmap of channels on which it receives, (1<<0)=ch 1...(1<<15)=ch 16.

```
CFStringRefkMIDIPropertyReceiveChannels;
```

```
kMIDIPropertyMaxSysExSpeed
```

Entity/endpoint property, integer. Set by the owning driver; should not be touched by other clients. Maximum bytes/second of sysex messages sent to it (default is 3125, as with MIDI 1.0)

```
CFStringRefkMIDIPropertyMaxSysExSpeed;
```

```
kMIDIPropertyAdvanceScheduleTimeMuSec
```

Device/entity/endpoint property, integer.

Set by the owning driver; should not be touched by other clients. If it is > 0, then it is a recommendation of how many microseconds in advance clients should schedule output. Clients should treat this value as a minimum. For devices with a > 0 advance schedule time, drivers will receive outgoing messages to the device at the time they are sent by the client, via MIDISend, and the driver is responsible for scheduling events to be played at the right times according to their timestamps.

```
CFStringRefkMIDIPropertyAdvanceScheduleTimeMuSec;
```

```
CFStringRefkMIDIPropertyIsEmbeddedEntity;
```

```
CFStringRefkMIDIPropertyConnectionUniqueID;

CFStringRefkMIDIPropertyDriverOwner;
```

Functions

MIDIClient

MIDIClientCreate

Creates a MIDIClient object.

```
MIDIClientCreate( CFStringRef name,   MIDINotifyProc notifyProc,   void *
                    notifyRefCon,   MIDIClientRef * outClient );
```

`name`	The client's name.
`notifyProc`	An optional (may be NULL) callback function through which the client will receive notifications of changes to the system.
`notifyRefCon`	A refCon passed back to notifyRefCon.
`outClient`	On successful return, points to the newly-created MIDIClientRef.

DISCUSSION

Result: An OSStatus result code.

MIDIClientDispose

Disposes a MIDIClient object.

```
MIDIClientDispose( MIDIClientRef client );
```

client The client to dispose.

DISCUSSION

Result: An OSStatus result code.

MIDIPort

MIDIInputPortCreate

Creates an input port through which the client may receive incoming MIDI messages from any MIDI source.

```
MIDIInputPortCreate( MIDIClientRef client, CFStringRef portName,
                     MIDIReadProc readProc, void * refCon, MIDIPortRef *
                     outPort);
```

`client` The client to own the newly-created port.

`portName` The name of the port.

`readProc` The MIDIReadProc which will be called with incoming MIDI, from sources connected to this port.

`refCon` The refCon passed to readHook.

`outPort` On successful return, points to the newly-created MIDIPort.

DISCUSSION

After creating a port, use MIDIPortConnectSource to establish an input connection from any number of sources to your port.

Result: An OSStatus result code.

MIDIOutputPortCreate

Creates an output port through which the client may send outgoing MIDI messages to any MIDI destination.

```
MIDIOutputPortCreate( MIDIClientRef client,   CFStringRef portName,
                      MIDIPortRef * outPort );
```

client	The client to own the newly-created port
portName	The name of the port.
outPort	On successful return, points to the newly-created MIDIPort.

DISCUSSION

Output ports provide a mechanism for MIDI merging. The system assumes that each output port will be responsible for sending only a single MIDI stream to each destination, although a single port may address all of the destinations in the system.

Result: An OSStatus result code.

MIDIPortDispose

Disposes a MIDIPort object.

```
MIDIPortDispose( MIDIPortRef port );
```

port	The port to dispose.

DISCUSSION

It is not usually necessary to call this function; when an application's MIDIClient's are automatically disposed at termination, or explicitly, via MIDIClientDispose, the client's ports are automatically disposed at this time.

Result: An OSStatus result code.

MIDIPortConnectSource

Establishes a connection from a source to a client's input port.

```
MIDIPortConnectSource( MIDIPortRef port, MIDIEndpointRef source, void *
                    connRefCon );
```

`port`	The port to which to create the connection. This port's readProc is called with incoming MIDI from the source.
`source`	The source from which to create the connection.
`connRefCon`	This refCon is passed to the MIDIReadProc, as a way to identify the source.

DISCUSSION

Result: An OSStatus result code.

MIDIPortDisconnectSource

Closes a previously-established source-to-input port connection.

```
MIDIPortDisconnectSource( MIDIPortRef port, MIDIEndpointRef source );
```

`port`	The port whose connection is being closed.
`source`	The source from which to close a connection to the specified port.

DISCUSSION

Result: An OSStatus result code.

System Information

MIDIGetNumberOfDevices

Returns the number of devices in the system.

```
MIDIGetNumberOfDevices();
```

DISCUSSION

Result: The number of devices in the system, or 0 if an error occurred.

MIDIGetDevice

Returns one of the devices in the system.

```
MIDIDeviceRef MIDIGetDevice( ItemCount deviceIndex0 );
```

`deviceIndex0` The index (0...MIDIGetNumberOfDevices()-1) of the device to return.

DISCUSSION

Use this to enumerate the devices in the system.

To enumerate the entities in the system, you can walk through the devices, then walk through the devices' entities.

Note: If a client iterates through the devices and entities in the system, it will never visit any virtual sources and destinations created by other clients. Also, a device iteration will return devices which are "offline" (those that were present in the past but are not currently present), while iterations through the system's sources and destinations will not include the endpoints of offline devices.

Thus, clients should usually prefer `MIDIGetNumberOfSources`, MIDIGetSource, `MIDIGetNumberOfDestinations` and `MIDIGetDestination` to iterating through devices and entities to locate endpoints.

Result: A reference to a device, or NULL if an error occurred.

MIDIGetNumberOfSources

Returns the number of sources in the system.

```
MIDIGetNumberOfSources();
```

DISCUSSION

Result: The number of sources in the system, or 0 if an error occurred.

MIDIGetSource

Returns one of the sources in the system.

```
MIDIGetSource( ItemCount sourceIndex0 );
```

`sourceIndex0` The index (0...MIDIGetNumberOfSources()-1) of the source to return.

DISCUSSION

Result: A reference to a source, or NULL if an error occurred.

MIDIGetNumberOfDestinations

Returns the number of destinations in the system.

```
extern ItemCount MIDIGetNumberOfDestinations();
```

DISCUSSION

Result: The number of destinations in the system, or 0 if an error occurred.

MIDIGetDestination

Returns one of the destinations in the system.

```
extern MIDIEndpointRef MIDIGetDestination( ItemCount destIndex0 );
```

`destIndex0` The index (0...MIDIGetNumberOfDestinations()-1) of the destination to return.

DISCUSSION

Result: A reference to a destination, or NULL if an error occurred.

Virtual Endpoints

MIDIDestinationCreate

Creates a virtual destination in a client.

```
MIDIDestinationCreate( MIDIClientRef client, CFStringRef name,
                       MIDIReadProc readProc, void * refCon,
                       MIDIEndpointRef * outDest );
```

`client` The client owning the virtual destination.

`name` The name of the virtual destination.

`readProc` The MIDIReadProc to be called when a client sends MIDI to the virtual destination.

`refCon` The refCon to be passed to the readProc.

`outDest` On successful return, a pointer to the newly-created destination.

DISCUSSION

Clients may use this to create virtual destinations.

The specified readProc gets called when clients send MIDI to your virtual destination.

Drivers need not call this; when they create devices and entities, sources and destinations are created at that time.

Result: An OSStatus result code.

MIDISourceCreate

Creates a virtual source in a client.

```
MIDISourceCreate( MIDIClientRef client, CFStringRef name,
                  MIDIEndpointRef * outSrc );
```

`client`	The client owning the virtual source.
`name`	The name of the virtual source.
`outSrc`	On successful return, a pointer to the newly-created source.

DISCUSSION

Clients may use this to create virtual sources.

Drivers need not call this; when they create devices and entities, sources and destinations are created at that time.

After creating a virtual source, use MIDIReceived to transmit MIDI messages from your virtual source to any clients connected to the virtual source.

Result: An OSStatus result code.

MIDIEndpointDispose

Disposes a virtual source or destination your client created.

```
MIDIEndpointDispose( MIDIEndpointRef endpt );
```

`endpt` The endpoint to be disposed.

DISCUSSION

Result: An OSStatus result code.

I/O

MIDISend

Send MIDI to a destination.

```
MIDISend( MIDIPortRef port, MIDIEndpointRef dest, const MIDIPacketList *
                    pktlist );
```

`port` The output port through which the MIDI is to be sent.

`dest` The destination to receive the events.

`pktlist` The MIDI events to be sent.

DISCUSSION

Events with future timestamps are scheduled for future delivery. The system performs any needed MIDI merging.

Result: An OSStatus result code.

MIDISendSysex

Send a single system-exclusive event, asynchronously.

```
MIDISendSysex( MIDISysexSendRequest *request );
```

`request` Contains the destination, and the MIDI data to be sent.

DISCUSSION

request->data must point to a single MIDI system-exclusive message, or portion thereof.

Result: An OSStatus result code.

MIDIReceived

Distributes MIDI from a source to the client input ports which are connected to that source.

```
MIDIReceived( MIDIEndpointRef src, const MIDIPacketList * pktlist );
```

`src` The source which is transmitting MIDI.

`pktlist` The MIDI events to be transmitted.

DISCUSSION

Drivers should call this function when receiving MIDI from a source.

Clients which have created virtual sources, using MIDISourceCreate, should call this function when the source is generating MIDI.

Result: An OSStatus result code.

MIDIObject

MIDIObjectGetIntegerProperty

Gets an object's integer-type property.

```
MIDIObjectGetIntegerProperty( MIDIObjectRef obj, CFStringRef propertyID,
                    SInt32 * outValue);
```

`obj`	The object whose property is to be returned.
`propertyID`	Name of the property to return.
`outValue`	On successful return, the value of the property.

DISCUSSION

See the MIDIObjectRef documentation for information about properties.)

Result: An OSStatus result code.

MIDIObjectSetIntegerProperty

Sets an object's integer-type property.

```
MIDIObjectSetIntegerProperty( MIDIObjectRef obj,   CFStringRef
                    propertyID,   SInt32 value );
```

`obj`	The object whose property is to be altered.
`propertyID`	Name of the property to set.
`value`	New value of the property.

DISCUSSION

See the MIDIObjectRef documentation for information about properties.

Result: An OSStatus result code.

MIDIObjectGetStringProperty

Gets an object's string-type property.

```
MIDIObjectGetStringProperty( MIDIObjectRef obj, CFStringRef propertyID,
                    CFStringRef * str );
```

`obj`	The object whose property is to be returned.
`propertyID`	Name of the property to return.
`str`	On successful return, the value of the property.

DISCUSSION

See the MIDIObjectRef documentation for information about properties.

Result: An OSStatus result code.

MIDIObjectSetStringProperty

Sets an object's string-type property.

```
MIDIObjectSetStringProperty(MIDIObjectRef obj,   CFStringRef propertyID,
                    CFStringRef str);
```

`obj`	The object whose property is to be altered.
`propertyID`	Name of the property to set.
`str`	New value of the property.

DISCUSSION

See the MIDIObjectRef documentation for information about properties.

Result: An OSStatus result code.

MIDIObjectGetDataProperty

Gets an object's data-type property.

```
MIDIObjectGetDataProperty( MIDIObjectRef obj, CFStringRef propertyID,
                    CFDataRef * outData );
```

`obj`	The object whose property is to be returned.
`propertyID`	Name of the property to return.
`outData`	On successful return, the value of the property.

DISCUSSION

See the MIDIObjectRef documentation for information about properties.)

Result: An OSStatus result code.

MIDIObjectSetDataProperty

Sets an object's data-type property.

```
MIDIObjectSetDataProperty( MIDIObjectRef obj,   CFStringRef propertyID,
                    CFDataRef data );
```

`obj`	The object whose property is to be altered.
`propertyID`	Name of the property to set.
`data`	New value of the property.

DISCUSSION

See the MIDIObjectRef documentation for information about properties.)

Result: An OSStatus result code.

MIDIDevice

MIDIDeviceGetNumberOfEntities

Returns the number of entities in a given device.

```
MIDIDeviceGetNumberOfEntities( MIDIDeviceRef device );
```

`device` The device being queried.

DISCUSSION

Result: The number of entities the device contains, or 0 if an error occurred.

MIDIEntity

MIDIEntityGetNumberOfSources

Returns the number of sources in a given entity.

```
MIDIEntityGetNumberOfSources( MIDIEntityRef entity );
```

`entity` The entity being queried.

DISCUSSION

Result: The number of sources the entity contains, or 0 if an error occurred.

MIDIEntityGetSource

Returns one of a given entity's sources.

```
MIDIEndpointRef MIDIEntityGetSource( MIDIEntityRef entity,   ItemCount
                    sourceIndex0 );
```

`entity` The entity being queried.

`sourceIndex0` The index (0...MIDIEntityGetNumberOfSources(entity)-1) of the source to return.

DISCUSSION

Result: A reference to a source, or NULL if an error occurred.

MIDIEntityGetNumberOfDestinations

Returns the number of destinations in a given entity.

```
MIDIEntityGetNumberOfDestinations( MIDIEntityRef entity );
```

`entity` The entity being queried.

DISCUSSION

Result: The number of destinations the entity contains, or 0 if an error occurred.

MIDIEntityGetDestination

Returns one of a given entity's destinations.

```
MIDIEntityGetDestination( MIDIEntityRef entity,  ItemCount destIndex0 );
```

`entity` The entity being queried.

`destIndex0`	The index (0...MIDIEntityGetNumberOfDestinations(entity) - 1) of the destination to return.

DISCUSSION

Result: A reference to a destination, or NULL if an error occurred.

MIDIDeviceGetEntity

Returns one of a given device's entities.

```
MIDIEntityRef MIDIDeviceGetEntity( MIDIDeviceRef device,  ItemCount
                    entityIndex0 );
```

`device`	The device being queried.
`entityIndex0`	The index (0...MIDIDeviceGetNumberOfEntities(device)-1) of the entity to return.

DISCUSSION

Result: A reference to an entity, or NULL if an error occurred.

MIDIPacketList Utilities

MIDIPacketNext

Advances a MIDIPacket pointer to the MIDIPacket which immediately follows it in memory if it is part of a MIDIPacketList.

```
MIDIPacket * MIDIPacketNext( MIDIPacket *pkt );
```

`pkt`	A pointer to a MIDIPacket in a MIDIPacketList.

DISCUSSION

This is implemented as a macro for efficiency and to avoid const problems.

Result: The subsequent packet.

MIDIPacketListInit

Prepares a MIDIPacketList to be built up dynamically.

```
MIDIPacket * MIDIPacketListInit( MIDIPacketList *pktlist );
```

`pktlist` The packet list to be initialized.

DISCUSSION

Result: A pointer to the first MIDIPacket in the packet list.

MIDIPacketListAdd

Adds a MIDI event to a MIDIPacketList.

```
MIDIPacket * MIDIPacketListAdd( MIDIPacketList * pktlist,  ByteCount
                    listSize,  MIDIPacket * curPacket,  MIDITimeStamp
                    time,  ByteCount nData,  Byte * data);
```

`pktlist` The packet list to which the event is to be added.

`listSize` The size, in bytes, of the packet list.

`curPacket` A packet pointer returned by a previous call to MIDIPacketListInit or MIDIPacketListAdd for this packet list.

`time` The new event's time.

`nData` The length of the new event, in bytes.

`data` The new event. May be a single MIDI event, or a partial sys-ex event. Running status is not permitted.

DISCUSSION

Result: Returns null if there wasn't room in the packet for the event; otherwise, returns a packet pointer which should be passed as `curPacket` in a subsequent call to this function.

Error Codes

```
enum {
    kMIDIInvalidClient      = -10830,
    kMIDIInvalidPort        = -10831,
    kMIDIWrongEndpointType  = -10832,       // want source, got
                                            // destination, or vice versa
    kMIDINoConnection       = -10833,       // attempt to close a
                                            // non-existent connection
    kMIDIUnknownEndpoint    = -10834,
    kMIDIUnknownProperty    = -10835,
    kMIDIWrongPropertyType  = -10836,
    kMIDINoCurrentSetup     = -10837,       // there is no current setup,
                                            // or it contains no devices
    kMIDIMessageSendErr     = -10838,       // communication with server
                                            // failed
    kMIDIServerStartErr     = -10839,       // couldn't start the server
    kMIDISetupFormatErr     = -10840,       // unparseable saved state
    kMIDIWrongThread        = -10841        // driver is calling non I/O
                                            // function in server than
                                            // from a thread other
                                            // server's main one
};
```

Core Audio Utilities

This reference chapter describes the shared Core Audio utilities available on Mac OS X.

The CoreAudioTypes API

The `CoreAudioTypes.h` file contains general structures and typedefs that are found and used throughout the audio and MIDI systems, including structures that represent an audio buffer, a structure describing the particular format of an audio stream, and structures for timing information.

The Host Time API

The APIs available in `HostTime.h` provide utility functions for converting between "host-time" and real-time, as expressed in nanoseconds.

Host time is the highest resolution clock that you can use on the system. On Mac OS X, this is the PowerPC decrementer register, which is the same counter that drives the Kernel Scheduler. Host time refers to values of numbers that correspond to that timebase. The `HostTime.h` file is comprised of routines to access information that include getting the current time and converting the hosttime to a useful time in nanoseconds.

Types

The following two structures are used to wrap up buffers of audio data when passing them around in API calls.

```
struct AudioBuffer
{
    UInt32  mNumberChannels; //number of interleaved channels in the
```

```
                            // buffer
    UInt32  mDataByteSize;  // the size of the buffer pointed to by
                            // mData
    void*   mData;          // the pointer to the buffer
};

typedef struct AudioBufferAudioBuffer;

struct AudioBufferList
{
    UInt32
    mNumberBuffers;
    AudioBuffer mBuffers[1];
};
typedef struct AudioBufferList  AudioBufferList;
```

This structure encapsulates all the information for describing the basic properties of a stream of audio data. The structure is sufficient to describe any constant bit rate format that has channels which are the same size. Extensions are required for variable bit rate data and for constant bit rate data where the channels have unequal sizes. However, where applicable, the appropriate fields will be filled out correctly for these kinds of formats (the extra data is provided via separate properties). In all fields, a value of 0 indicates that the field is either unknown, not applicable or otherwise is inapproprate for the format and should be ignored.

The extended description data, if applicable, is available via a property with the same ID as the format ID. The contents of the data are specific to the format.

```
struct AudioStreamBasicDescription
{
    Float64 mSampleRate;        //  the native sample rate of the audio
                                //  stream
    UInt32  mFormatID;          //  the specific encoding type of audio
                                //  stream
    UInt32  mFormatFlags;       //  flags specific to each format
    UInt32  mBytesPerPacket;    //  the number of bytes in a packet
    UInt32  mFramesPerPacket;   //  the number of frames in each packet
    UInt32  mBytesPerFrame;     //  the number of bytes in a frame
    UInt32  mChannelsPerFrame;  //  the number of channels in each frame
```

```
    UInt32  mBitsPerChannel;    //  the number of bits in each channel
};
typedef struct AudioStreamBasicDescription  AudioStreamBasicDescription;
```

The following struct is for encapsulating the parts of a time stamp. The flags define which fields are valid.

```
struct AudioTimeStamp
{
    Float64         mSampleTime;    //  the absolute sample time
    UInt64          mHostTime;      //  the host's root timebase's time
    Float64         mRateScalar;    //  the system rate scalar
    UInt64          mWordClockTime; //  the word clock time
    SMPTETime       mSMPTETime;     //  the SMPTE time
    UInt32          mFlags;         //  the flags indicate which fields
are valid
};
typedef struct AudioTimeStamp   AudioTimeStamp;
```

The following is a struct for encapsulating a SMPTE time. The running rate should be expressed in the AudioTimeStamp's `mRateScalar` field.

```
struct SMPTETime
{
    UInt64  mCounter;           //  total number of messages received
    UInt32  mType;              //  the SMPTE type (see constants)
    UInt32  mFlags;             //  flags indicating state (see constants
    SInt16  mHours;             //  number of hours in the full message
    SInt16  mMinutes;           //  number of minutes in the full message
    SInt16  mSeconds;           //  number of seconds in the full message
    SInt16  mFrames;            //  number of frames in the full message
};
typedef struct SMPTETime    SMPTETime;
```

Constants

The following constants describe SMPTE types (taken from the MTC spec).

```
enum
{
    kSMPTETimeType24          = 0,
    kSMPTETimeType25         = 1,
    kSMPTETimeType30Drop      = 2,
    kSMPTETimeType30          = 3,
    kSMPTETimeType2997        = 4,
    kSMPTETimeType2997Drop    = 5
};
```

The following flags describe a SMPTE time stamp.

```
enum
{
    kSMPTETimeValid = (1L << 0),    //  the full time is valid
    kSMPTETimeRunning = (1L << 1)       //  time is running
};
```

The following flags are for the `AudioTimeStamp` stucture.

```
enum
{
    kAudioTimeStampSampleTimeValid       = (1L << 0),
    kAudioTimeStampHostTimeValid         = (1L << 1),
    kAudioTimeStampRateScalarValid      = (1L << 2),
    kAudioTimeStampWordClockTimeValid   = (1L << 3),
    kAudioTimeStampSMPTETimeValid        = (1L << 4)
};
```

The following flag is used for the `AudioFormatLinearPCM` stucture.

```
enum
{
    kAudioFormatLinearPCM    = 'lpcm'
};
```

The following flags are used in the `mFormatFlags` field of `AudioStreamBasicDescription` to describe linear PCM data.

```
enum{
```

`kLinearPCMFormatFlagIsFloat = (1L << 0)`

Set this for floating point and clear for integer.

`kLinearPCMFormatFlagIsBigEndian = (1L << 1)`

Set for big endian and clear for little endian.

`kLinearPCMFormatFlagIsSignedInteger = (1L << 2)`

Set for signed integer, clear for unsigned integer; only valid if kLinearPCMFormatFlagIsFloat is clear.

`kLinearPCMFormatFlagIsPacked = (1L << 3)`

Set this if the sample bits are packed as closely together as possible; clear if they are high or low aligned within the channel.

`kLinearPCMFormatFlagIsAlignedHigh = (1L << 4)`

Set if the sample bits are placed into the high bits of the channel, clear for low bit placement; only valid if kLinearPCMFormatFlagIsPacked is clear.

```
};
```

The following constants are for use in AudioStreamBasicDescriptions.

```
enum {
    kAudioStreamAnyRate = 0
};
```

The format can use any sample rate (usually because it does its own rate conversion). Note that this constant can only appear in listings of supported descriptions. It should never appear in the current description as a device must always have a "current" nominal sample rate.

Host Time

The following are routines for accessing the host's time base.

AudioGetCurrentHostTime

Retrieves the current host time value.

```
AudioGetCurrentHostTime();
```

AudioGetHostClockFrequency

Retrieves the number of ticks per second of the host clock.

```
AudioGetHostClockFrequency();
```

AudioGetHostClockMinimumTimeDelta

Retrieves the smallest number of ticks difference between two succeeding values of the host clock. For instance, if this value is 5 and the first value of the host clock is X, then the next time after X will be at greater than or equal to X+5.

```
AudioGetHostClockMinimumTimeDelta();
```

AudioConvertHostTimeToNanos

Converts the given host time to a time in nanoseconds.

```
AudioConvertHostTimeToNanos(UInt64 inHostTime);
```

AudioConvertNanosToHostTime

Converts the given nanoseconds time to a time in the host clock's time base.

```
AudioConvertNanosToHostTime(UInt64 inNanos);
```

A

D

M

N

Document Revision History

The following is a change log of this document, which introduced the new core audio and MIDI software architecture to developers at Apple's World Wide Developer Conference in May, 2001.

Table A-1 Audio and MIDI on Mac OS X revision history

Version	Notes
05/18/01	Alpha draft of document completed, based on input from the core audio engineering team at Apple. PDF generated for engineering review.
05/22/01	Revised document to include engineering changes and updates.
05/29/01	Reorganized chapters to include both overview and reference material. Regrouped types, constants in reference sections. Added types for CoreMIDI chapter from .h file.

APPENDIX A

Document Revision History