Audio Unit Hosting Guide for iOS

Developer

Contents

About Audio Unit Hosting 6
At a Glance 6
Audio Units Provide Fast, Modular Audio Processing 7
Choosing a Design Pattern and Constructing Your App 7
Get the Most Out of Each Audio Unit 8
How to Use This Document 9
Prerequisites 9
See Also 9

Audio Unit Hosting Fundamentals 10
Audio Units Provide Fast, Modular Audio Processing 10
Audio Units in iOS 10
Use the Two Audio Unit APIs in Concert 12
Use Identifiers to Specify and Obtain Audio Units 13
Use Scopes and Elements to Specify Parts of Audio Units 15
Use Properties to Configure Audio Units 16
Use Parameters and UIKit to Give Users Control 18
Essential Characteristics of I/O Units 18
Audio Processing Graphs Manage Audio Units 20
An Audio Processing Graph Has Exactly One I/O Unit 21
Audio Processing Graphs Provide Thread Safety 21
Audio Flows Through a Graph Using "Pull" 23
Render Callback Functions Feed Audio to Audio Units 24
Understanding the Audio Unit Render Callback Function 25
Audio Stream Formats Enable Data Flow 27
Working with the AudioStreamBasicDescription structure 27
Understanding Where and How to Set Stream Formats 29

Constructing Audio Unit Apps 32
Start by Choosing a Design Pattern 32
I/O Pass Through 33
I/O Without a Render Callback Function 34
I/O with a Render Callback Function 34
Output-Only with a Render Callback Function 36

Other Audio Unit Hosting Design Patterns 37
Constructing Your App 37
Configure Your Audio Session 38
Specify the Audio Units You Want 39
Build an Audio Processing Graph 39
Configure the Audio Units 40
Write and Attach Render Callback Functions 41
Connect the Audio Unit Nodes 42
Provide a User Interface 43
Initialize and Start the Audio Processing Graph 44
Troubleshooting Tips 44

Using Specific Audio Units 46
Using I/O Units 46
Remote I/O Unit 46
Voice-Processing I/O Unit 47
Generic Output Unit 48
Using Mixer Units 48
Multichannel Mixer Unit 48
3D Mixer Unit 49
Using Effect Units 50
Identifier Keys for Audio Units 52

Document Revision History 53

Figures, Tables, and Listings

Audio Unit Hosting Fundamentals 10

Figure 1-1 Audio frameworks in iOS 10
Figure 1-2 Audio unit scopes and elements 15
Figure 1-3 The architecture of an I/O unit 19
Figure 1-4 A simple audio processing graph for playback 22
Figure 1-5 The same graph after inserting an equalizer 22
Figure 1-6 The pull mechanism of audio data flow 23
Figure 1-7 The `ioData` buffers for a stereo render callback function 26
Figure 1-8 Where to set audio data stream formats 30
Table 1-1 Audio units provided in iOS 11
Listing 1-1 Creating an audio component description to identify an audio unit 13
Listing 1-2 Obtaining an audio unit instance using the audio unit API 13
Listing 1-3 Obtaining an audio unit instance using the audio processing graph API 14
Listing 1-4 Using scope and element when setting a property 17
Listing 1-5 A render callback function header 25
Listing 1-6 The `AudioStreamBasicDescription` structure 27
Listing 1-7 Defining an ASBD for a stereo stream 27

Constructing Audio Unit Apps 32

Figure 2-1 Simultaneous I/O pass through 33
Figure 2-2 Simultaneous I/O without a render callback function 34
Figure 2-3 Simultaneous I/O with a render callback function 35
Figure 2-4 Output-only with a render callback function 36
Figure 2-5 A more complex example of output-only with a render callback function 36
Listing 2-1 Configuring an audio session 38
Listing 2-2 Building an audio processing graph 39
Listing 2-3 Attaching a render callback immediately 41
Listing 2-4 Attaching a render callback in a thread-safe manner 42
Listing 2-5 Connecting two audio unit nodes using the audio processing graph API 42
Listing 2-6 Connecting two audio units directly 43
Listing 2-7 Initializing and starting an audio processing graph 44
Listing 2-8 A utility method to print field values for an `AudioStreamBasicDescription` structure 45

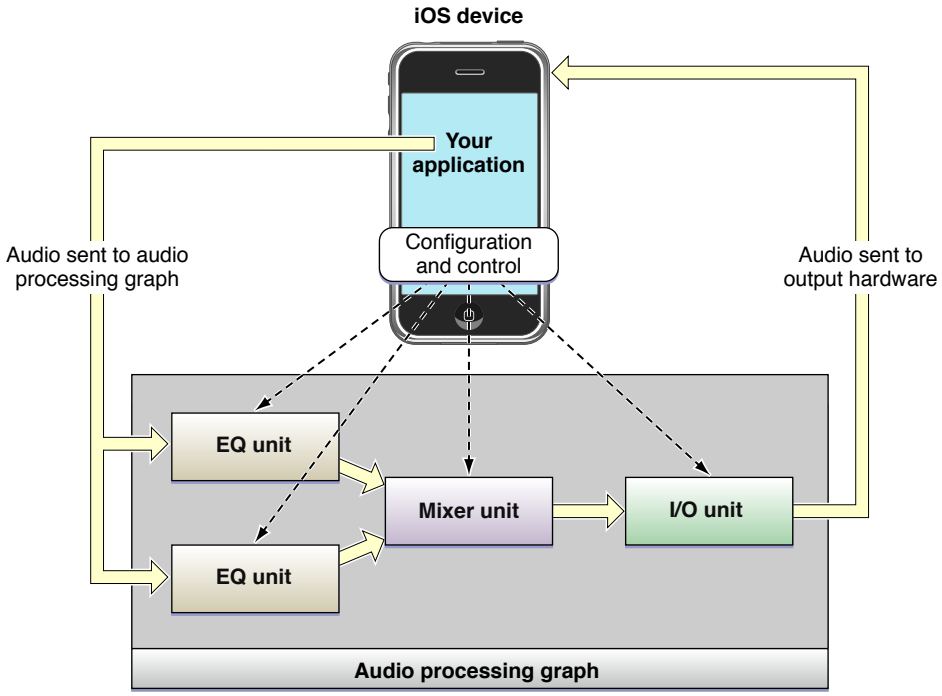
Using Specific Audio Units 46

Table 3-1 Using the Remote I/O unit 47
Table 3-2 Using the Multichannel Mixer unit 48
Table 3-3 Using the 3D Mixer unit 50
Table 3-4 Using the iPod EQ unit 51
Table 3-5 Identifier keys for accessing the dynamically-linkable libraries for each iOS audio unit 52

About Audio Unit Hosting

iOS provides audio processing plug-ins that support mixing, equalization, format conversion, and realtime input/output for recording, playback, offline rendering, and live conversation such as for VoIP (Voice over Internet Protocol). You can dynamically load and use—that is, *host* —these powerful and flexible plug-ins, known as *audio units* , from your iOS application.

Audio units usually do their work in the context of an enclosing object called an *audio processing graph* , as shown in the figure. In this example, your app sends audio to the first audio units in the graph by way of one or more callback functions and exercises individual control over each audio unit. The output of the I/O unit—the last audio unit in this or any audio processing graph—connects directly to the output hardware.



At a Glance

Because audio units constitute the lowest programming layer in the iOS audio stack, to use them effectively requires deeper understanding than you need for other iOS audio technologies. Unless you require realtime playback of synthesized sounds, low-latency I/O (input and output), or specific audio unit features, look first

at the Media Player, AV Foundation, OpenAL, or Audio Toolbox frameworks. These higher-level technologies employ audio units on your behalf and provide important additional features, as described in *Multimedia Programming Guide*.

Audio Units Provide Fast, Modular Audio Processing

The two greatest advantages of using audio units directly are:

- Excellent responsiveness. Because you have access to a realtime priority thread in an audio unit render callback function, your audio code is as close as possible to the metal. Synthetic musical instruments and realtime simultaneous voice I/O benefit the most from using audio units directly.
- Dynamic reconfiguration. The audio processing graph API, built around the `AUGraph` opaque type, lets you dynamically assemble, reconfigure, and rearrange complex audio processing chains in a thread-safe manner, all while processing audio. This is the only audio API in iOS offering this capability.

An audio unit's life cycle proceeds as follows:

1. At runtime, obtain a reference to the dynamically-linkable library that defines an audio unit you want to use.
2. Instantiate the audio unit.
3. Configure the audio unit as required for its type and to accomodate the intent of your app.
4. Initialize the audio unit to prepare it to handle audio.
5. Start audio flow.
6. Control the audio unit.
7. When finished, deallocate the audio unit.

Audio units provide highly useful individual features such as stereo panning, mixing, volume control, and audio level metering. Hosting audio units lets you add such features to your app. To reap these benefits, however, you must gain facility with a set of fundamental concepts including audio data stream formats, render callback functions, and audio unit architecture.

Relevant Chapter: "Audio Unit Hosting Fundamentals" (page 10)

Choosing a Design Pattern and Constructing Your App

An audio unit hosting design pattern provides a flexible blueprint to customize for the specifics of your app. Each pattern indicates:

- How to configure the I/O unit. I/O units have two independent elements, one that accepts audio from the input hardware, one that sends audio to the output hardware. Each design pattern indicates which element or elements you should enable.
- Where, within the audio processing graph, you must specify audio data stream formats. You must correctly specify formats to support audio flow.
- Where to establish audio unit connections and where to attach your render callback functions. An audio unit connection is a formal construct that propagates a stream format from an output of one audio unit to an input of another audio unit. A render callback lets you feed audio into a graph or manipulate audio at the individual sample level within a graph.

No matter which design pattern you choose, the steps for constructing an audio unit hosting app are basically the same:

1. Configure your application audio session to ensure your app works correctly in the context of the system and device hardware.
2. Construct an audio processing graph. This multistep process makes use of everything you learned in "Audio Unit Hosting Fundamentals" (page 10).
3. Provide a user interface for controlling the graph's audio units.

Become familiar with these steps so you can apply them to your own projects.

Relevant Chapter: "Constructing Audio Unit Apps" (page 32)

Get the Most Out of Each Audio Unit

Most of this document teaches you that all iOS audio units share important, common attributes. These attributes include, for example, the need for your app to specify and load the audio unit at runtime, and then to correctly specify its audio stream formats.

At the same time, each audio unit has certain unique features and requirements, ranging from the correct audio sample data type to use, to required configuration for correct behavior. Understand the usage details and specific capabilities of each audio unit so you know, for example, when to use the 3D Mixer unit and when to instead use the Multichannel Mixer.

Relevant Chapter: "Using Specific Audio Units" (page 46)

How to Use This Document

If you prefer to begin with a hands-on introduction to audio unit hosting in iOS, download one of the sample apps available in the iOS Dev Center, such as *Audio Mixer (MixerHost)*. Come back to this document to answer questions you may have and to learn more.

If you want a solid conceptual grounding before starting your project, read "Audio Unit Hosting Fundamentals" (page 10) first. This chapter explains the concepts behind the APIs. Continue with "Constructing Audio Unit Apps" (page 32) to learn about picking a design pattern for your project and the workflow for building your app.

If you have some experience with audio units and just want the specifics for a given type, you can start with "Using Specific Audio Units" (page 46).

Prerequisites

Before reading this document, it's a good idea to read the section "A Little About Digital Audio and Linear PCM" in *Core Audio Overview*. Also, review *Core Audio Glossary* for terms you may not already be familiar with. To check if your audio needs might be met by a higher-level technology, review "Using Audio" in *Multimedia Programming Guide*.

See Also

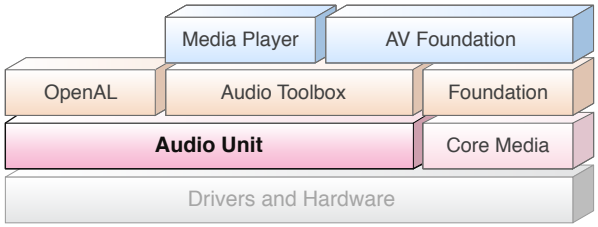
Essential reference documentation for building an audio unit hosting app includes the following:

- *Audio Unit Properties Reference* describes the properties you can use to configure each type of audio unit.
- *Audio Unit Parameters Reference* describes the parameters you can use to control each type of audio unit.
- *Audio Unit Component Services Reference* describes the API for accessing audio unit parameters and properties, and describes the various audio unit callback functions.
- *Audio Component Services Reference* describes the API for accessing audio units at runtime and for managing audio unit instances.
- *Audio Unit Processing Graph Services Reference* describes the API for constructing and manipulating audio processing graphs, which are dynamically reconfigurable audio processing chains.
- *Core Audio Data Types Reference* describes the data structures and types you need for hosting audio units.

Audio Unit Hosting Fundamentals

All audio technologies in iOS are built on top of audio units, as shown in Figure 1-1. The higher-level technologies shown here—Media Player, AV Foundation, OpenAL, and Audio Toolbox—wrap audio units to provide dedicated and streamlined APIs for specific tasks.

Figure 1-1 Audio frameworks in iOS



Direct use of audio units in your project is the correct choice only when you need the very highest degree of control, performance, or flexibility, or when you need a specific feature (such as acoustic echo cancelation) available only by using an audio unit directly. For an overview of iOS audio APIs, and guidance on when to use each one, refer to *Multimedia Programming Guide*.

Audio Units Provide Fast, Modular Audio Processing

Use audio units directly, rather than by way of higher-level APIs, when you require one of the following:

- Simultaneous audio I/O (input and output) with low latency, such as for a VoIP (Voice over Internet Protocol) application
- Responsive playback of synthesized sounds, such as for musical games or synthesized musical instruments
- Use of a specific audio unit feature such as acoustic echo cancelation, mixing, or tonal equalization
- A processing-chain architecture that lets you assemble audio processing modules into flexible networks. This is the only audio API in iOS offering this capability.

Audio Units in iOS

iOS provides seven audio units arranged into four categories by purpose, as shown in Table 1-1.

Table 1-1 Audio units provided in iOS

Purpose	Audio units
Effect	iPod Equalizer
Mixing	3D Mixer Multichannel Mixer
I/O	Remote I/O Voice-Processing I/O Generic Output
Format conversion	Format Converter

The identifiers you use to specify these audio units programmatically are listed in "Identifier Keys for Audio Units" (page 52).

> **Note:** The iOS dynamic plug-in architecture does not support third-party audio units. That is, the only audio units available for dynamic loading are those provided by the operating system.

Effect Unit

iOS 4 provides one effect unit, the **iPod Equalizer**, the same equalizer used by the built-in iPod app. To view the iPod app's user interface for this audio unit, go to Settings > iPod > EQ. When using this audio unit, you must provide your own UI. This audio unit offers a set of preset equalization curves such as Bass Booster, Pop, and Spoken Word.

Mixer Units

iOS provides two mixer units. The **3D Mixer unit** is the foundation upon which OpenAL is built. In most cases, if you need the features of the 3D Mixer unit, your best option is to use OpenAL which provides a higher level API that is well suited for game apps. For sample code that shows how to use OpenAL, see the sample code project *oalTouch*.

The **Multichannel Mixer unit** provides mixing for any number of mono or stereo streams, with a stereo output. You can turn each input on or off, set its input gain, and set its stereo panning position. For a demonstration of how to use this audio unit, see the sample code project *Audio Mixer (MixerHost)*.

I/O Units

iOS provides three I/O units. The **Remote I/O unit** is the most commonly used. It connects to input and output audio hardware and gives you low–latency access to individual incoming and outgoing audio sample values. It provides format conversion between the hardware audio formats and your application audio format, doing so by way of an included Format Converter unit. For sample code that shows how to use the Remote I/O unit, see the sample code project *aurioTouch*.

The **Voice-Processing I/O unit** extends the Remote I/O unit by adding acoustic echo cancelation for use in a VoIP or voice-chat application. It also provides automatic gain correction, adjustment of voice-processing quality, and muting.

The **Generic Output unit** does not connect to audio hardware but rather provides a mechanism for sending the output of a processing chain to your application. You would typically use the Generic Output unit for offline audio processing.

Format Converter Unit

iOS 4 provides one **Format Converter unit**, which is typically used indirectly by way of an I/O unit.

Use the Two Audio Unit APIs in Concert

iOS has one API for working with audio units directly and another for manipulating audio processing graphs. When you host audio units in your app, you use both APIs in concert.

- To work with audio units directly—configuring and controlling them—use the functions described in *Audio Unit Component Services Reference*.
- To create and configure an audio processing graph (a processing chain of audio units) use the functions described in *Audio Unit Processing Graph Services Reference*.

There is some overlap between the two APIs and you are free to mix and match according to your programming style. The audio unit API and audio processing graph API each provide functions for:

- Obtaining references to the dynamically-linkable libraries that define audio units
- Instantiating audio units
- Interconnecting audio units and attaching render callback functions
- Starting and stopping audio flow

This document provides code examples for using both APIs but focuses on the audio processing graph API. Where there is a choice between the two APIs in your code, use the processing graph API unless you have a specific reason not to. Your code will be more compact, easier to read, and more amenable to supporting dynamic reconfiguration (see "Audio Processing Graphs Provide Thread Safety" (page 21)).

Use Identifiers to Specify and Obtain Audio Units

To find an audio unit at runtime, start by specifying its type, subtype, and manufacturer keys in an audio component description data structure. You do this whether using the audio unit or audio processing graph API. Listing 1-1 shows how.

Listing 1-1 Creating an audio component description to identify an audio unit

```
AudioComponentDescription ioUnitDescription;

ioUnitDescription.componentType          = kAudioUnitType_Output;
ioUnitDescription.componentSubType       = kAudioUnitSubType_RemoteIO;
ioUnitDescription.componentManufacturer  = kAudioUnitManufacturer_Apple;
ioUnitDescription.componentFlags         = 0;
ioUnitDescription.componentFlagsMask     = 0;
```

This description specifies exactly one audio unit—the Remote I/O unit. The keys for this and other iOS audio units are listed in "Identifier Keys for Audio Units" (page 52). Note that all iOS audio units use the `kAudioUnitManufacturer_Apple` key for the `componentManufacturer` field.

To create a wildcard description, set one or more of the type/subtype fields to 0. For example, to match all the I/O units, change Listing 1-1 to use a value of 0 for the `componentSubType` field.

With a description in hand, you obtain a reference to the library for the specified audio unit (or set of audio units) using either of two APIs. The audio unit API is shown in Listing 1-2.

Listing 1-2 Obtaining an audio unit instance using the audio unit API

```
AudioComponent foundIoUnitReference = AudioComponentFindNext (
                                          NULL,
                                          &ioUnitDescription
                                      );
AudioUnit ioUnitInstance;
AudioComponentInstanceNew (
```

```
    foundIoUnitReference,
    &ioUnitInstance
);
```

Passing `NULL` to the first parameter of `AudioComponentFindNext` tells this function to find the first system audio unit matching the description, using a system-defined ordering. If you instead pass a previously found audio unit reference in this parameter, the function locates the next audio unit matching the description. This usage lets you, for example, obtain references to all of the I/O units by repeatedly calling `AudioComponentFindNext`.

The second parameter to the `AudioComponentFindNext` call refers to the audio unit description defined in Listing 1-1 (page 13).

The result of the `AudioComponentFindNext` function is a reference to the dynamically-linkable library that defines the audio unit. Pass the reference to the `AudioComponentInstanceNew` function to instantiate the audio unit, as shown in Listing 1-2 (page 13).

You can instead use the audio processing graph API to instantiate an audio unit. Listing 1-3 shows how.

Listing 1-3 Obtaining an audio unit instance using the audio processing graph API

```
// Declare and instantiate an audio processing graph
AUGraph processingGraph;
NewAUGraph (&processingGraph);

// Add an audio unit node to the graph, then instantiate the audio unit
AUNode ioNode;
AUGraphAddNode (
    processingGraph,
    &ioUnitDescription,
    &ioNode
);
AUGraphOpen (processingGraph); // indirectly performs audio unit instantiation

// Obtain a reference to the newly-instantiated I/O unit
AudioUnit ioUnit;
AUGraphNodeInfo (
```

```
    processingGraph,
    ioNode,
    NULL,
    &ioUnit
);
```

This code listing introduces `AUNode`, an opaque type that represents an audio unit in the context of an audio processing graph. You receive a reference to the new audio unit instance, in the `ioUnit` parameter, on output of the `AUGraphNodeInfo` function call.

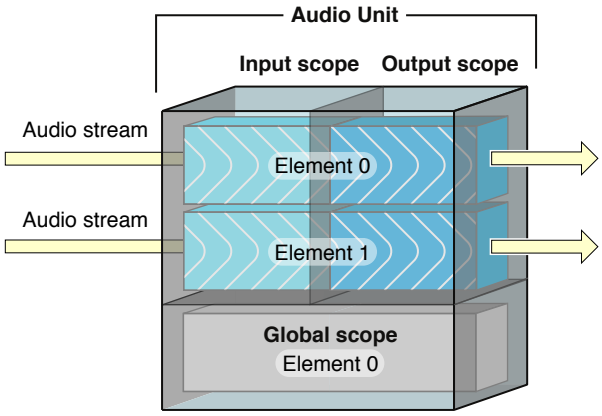
The second parameter to the `AUGraphAddNode` call refers to the audio unit description defined in Listing 1-1 (page 13).

Having obtained an audio unit instance, you can configure it. To do so, you need to learn about two audio unit characteristics, *scopes* and *elements*.

Use Scopes and Elements to Specify Parts of Audio Units

The parts of an audio unit are organized into scopes and elements, as shown in Figure 1-2. When invoking a function to configure or control an audio unit, you specify the scope and element to identify the specific target of the function.

Figure 1-2 Audio unit scopes and elements



A **scope** is a programmatic context within an audio unit. Although the name *global scope* might suggest otherwise, these contexts are never nested. You specify the scope you are targeting by using a constant from the `Audio Unit Scopes` enumeration.

An **element** is a programmatic context nested within an audio unit scope. When an element is part of an input or output scope, it is analogous to a signal bus in a physical audio device—and for that reason is sometimes called a bus. These two terms—*element* and *bus*—refer to exactly the same thing in audio unit programming. This document uses "bus" when emphasizing signal flow and uses "element" when emphasizing a specific functional aspect of an audio unit, such the input and output elements of an I/O unit (see "Essential Characteristics of I/O Units" (page 18)).

You specify an element (or bus) by its zero-indexed integer value. If setting a property or parameter that applies to a scope as a whole, specify an element value of 0.

Figure 1-2 (page 15) illustrates one common architecture for an audio unit, in which the numbers of elements on input and output are the same. However, various audio units use various architectures. A mixer unit, for example, might have several input elements but a single output element. You can extend what you learn here about scopes and elements to any audio unit, despite these variations in architecture.

The global scope, shown at the bottom of Figure 1-2 (page 15), applies to the audio unit as a whole and is not associated with any particular audio stream. It has exactly one element, namely element 0. Some properties, such as maximum frames per slice (`kAudioUnitProperty_MaximumFramesPerSlice`), apply only to the global scope.

The input and output scopes participate directly in moving one or more audio streams through the audio unit. As you'd expect, audio enters at the input scope and leaves at the output scope. A property or parameter may apply to an input or output scope as a whole, as is the case for the element count property (`kAudioUnitProperty_ElementCount`), for example. Other properties and parameters, such as the enable I/O property (`kAudioOutputUnitProperty_EnableIO`) or the volume parameter (`kMultiChannelMixerParam_Volume`), apply to a specific element within a scope.

Use Properties to Configure Audio Units

An **audio unit property** is a key-value pair you can use to configure an audio unit. The key for a property is a unique integer with an associated mnemonic identifier, such as `kAudioUnitProperty_MaximumFramesPerSlice = 14`. Apple reserves property keys from 0 through 63999. In Mac OS X, third-party audio units make use of keys above this range.

The value for each property is of a designated data type and has a designated read/write access, as described in *Audio Unit Properties Reference*. To set any property on any audio unit, you use one flexible function: `AudioUnitSetProperty`. Listing 1-4 shows a typical use of this function, with comments highlighting how to specify the scope and element as well as indicating the key and value for the property.

Listing 1-4 Using scope and element when setting a property

```
UInt32 busCount = 2;


OSStatus result = AudioUnitSetProperty (
    mixerUnit,
    kAudioUnitProperty_ElementCount    // the property key
    kAudioUnitScope_Input,             // the scope to set the property on
    0,                                 // the element to set the property on
    &busCount,                         // the property value
    sizeof (busCount
);
```

Here are a few properties you'll use frequently in audio unit development. Become familiar with each of these by reading its reference documentation and by exploring Apple's audio unit sample code projects such as *Audio Mixer (MixerHost)*:

- `kAudioOutputUnitProperty_EnableIO`, for enabling or disabling input or output on an I/O unit. By default, output is enabled but input is disabled.
- `kAudioUnitProperty_ElementCount`, for configuring the number of input elements on a mixer unit, for example.
- `kAudioUnitProperty_MaximumFramesPerSlice`, for specifying the maximum number of frames of audio data an audio unit should be prepared to produce in response to a render call. For most audio units, in most scenarios, you must set this property as described in the reference documentation. If you don't, your audio will stop when the screen locks.
- `kAudioUnitProperty_StreamFormat`, for specifying the audio stream data format for a particular audio unit input or output bus.

Most property values can be set only when an audio unit is uninitialized. Such properties are not intended to be changed by the user. Some, though, such as the `kAudioUnitProperty_PresentPreset` property of the iPod EQ unit, and the `kAUVoiceIOProperty_MuteOutput` property of the Voice-Processing I/O unit, *are* intended to be changed while playing audio.

To discover a property's availability, access its value, and monitor changes to its value, use the following functions:

- `AudioUnitGetPropertyInfo`—To discover whether a property is available; if it is, you are given the data size for its value and whether or not you can change the value
- `AudioUnitGetProperty`, `AudioUnitSetProperty`—To get or set the value of a property

- `AudioUnitAddPropertyListener`, `AudioUnitRemovePropertyListenerWithUserData`—To install or remove a callback function for monitoring changes to a property's value

Use Parameters and UIKit to Give Users Control

An **audio unit parameter** is a user-adjustable setting that can change while an audio unit is producing audio. Indeed, the intention of most parameters (such as volume or stereo panning position) is real-time adjustment of the processing that an audio unit is performing.

Like an audio unit property, an audio unit parameter is a key-value pair. The key is defined by the audio unit it applies to. It is always an enumeration constant, such as `kMultiChannelMixerParam_Pan = 2`, that is unique to the audio unit but not globally unique.

Unlike property values, every parameter value is of the same type: 32-bit floating point. The permissible range for a value, and the unit of measure that it represents, are determined by the audio unit's implementation of the parameter. These and other aspects of the parameters in iOS audio units are described in *Audio Unit Parameters Reference*.

To get or set a parameter value, use one of the following functions, which are fully described in *Audio Unit Component Services Reference*:

- `AudioUnitGetParameter`
- `AudioUnitSetParameter`

To allow users to control an audio unit, give them access to its parameters by way of a user interface. Start by choosing an appropriate class from UIKit framework to represent the parameter. For example, for an on/off feature, such as the Multichannel Mixer unit's `kMultiChannelMixerParam_Enable` parameter, you could use a `UISwitch` object. For a continuously varying feature, such as stereo panning position as provided by the `kMultiChannelMixerParam_Pan` parameter, you could use a `UISlider` object.

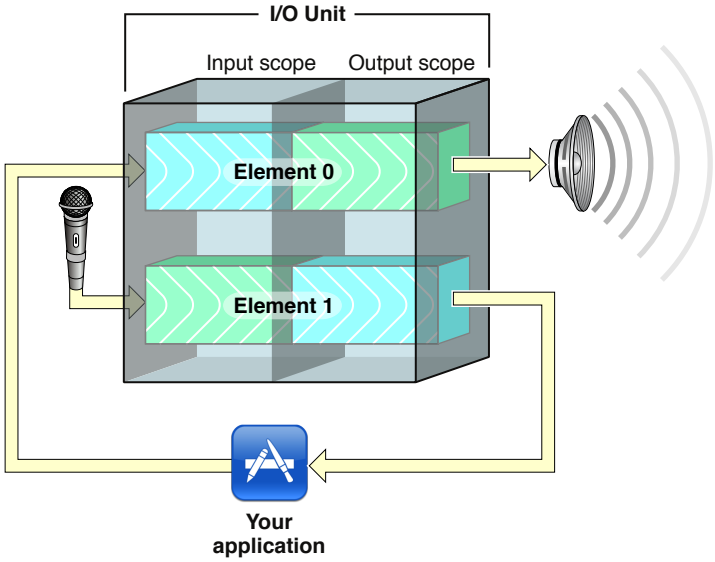
Convey the value of the UIKit object's current configuration—such as the position of the slider thumb for a `UISlider`—to the audio unit. Do so by wrapping the `AudioUnitSetParameter` function in an `IBAction` method and establishing the required connection in Interface Builder. For sample code illustrating how to do this, see the sample code project *Audio Mixer (MixerHost)*.

Essential Characteristics of I/O Units

I/O units are the one type of audio unit used in every audio unit app and are unusual in several ways. For both these reasons, you must become acquainted with the essential characteristics of I/O units to gain facility in audio unit programming.

An I/O unit contains exactly two elements, as you see in Figure 1-3.

Figure 1-3 The architecture of an I/O unit



Although these two elements are parts of one audio unit, your app treats them largely as independent entities. For example, you employ the enable I/O property (`kAudioOutputUnitProperty_EnableIO`) to enable or disable each element independently, according to the needs of your app.

Element 1 of an I/O unit connects directly to the audio input hardware on a device, represented in the figure by a microphone. This hardware connection—at the input scope of element 1—is opaque to you. Your first access to audio data entering from the input hardware is at the output scope of element 1.

Similarly, element 0 of an I/O unit connects directly the audio output hardware on a device, represented in Figure 1-3 by the loudspeaker. You can convey audio to the input scope of element 0, but its output scope is opaque.

Working with audio units, you'll often hear the two elements of an I/O unit described not by their numbers but by name:

- The **input element** is element 1 (mnemonic device: the letter "I" of the word "Input" has an appearance similar to the number 1)
- The **output element** is element 0 (mnemonic device: the letter "O" of the word "Output" has an appearance similar to the number 0)

As you see in Figure 1-3 (page 19), each element itself has an input scope and an output scope. For this reason, describing these parts of an I/O unit may get a bit confusing. For example, you would say that in a simultaneous I/O app, you receive audio from the output scope of the input element and send audio to the input scope of the output element. When you need to, return to this figure.

Finally, I/O units are the only audio units capable of starting and stopping the flow of audio in an audio processing graph. In this way, the I/O unit is in charge of the audio flow in your audio unit app.

Audio Processing Graphs Manage Audio Units

An **audio processing graph** is a Core Foundation–style opaque type, `AUGraph`, that you use to construct and manage an audio unit processing chain. A graph can leverage the capabilities of multiple audio units and multiple render callback functions, allowing you to create nearly any audio processing solution you can imagine.

The `AUGraph` type adds thread safety to the audio unit story: It enables you to reconfigure a processing chain on the fly. For example, you could safely insert an equalizer, or even swap in a different render callback function for a mixer input, while audio is playing. In fact, the `AUGraph` type provides the only API in iOS for performing this sort of dynamic reconfiguration in an audio app.

The audio processing graph API uses another opaque type, `AUNode`, to represent an individual audio unit within the context of a graph. When using a graph, you usually interact with nodes as proxies for their contained audio units rather than interacting with the audio units directly.

When putting a graph together, however, you must configure each audio unit, and to do that you must interact directly with the audio units by way of the audio unit API. Audio unit nodes, per se, are not configurable. In this way, constructing a graph requires you to use the both APIs, as explained in "Use the Two Audio Unit APIs in Concert" (page 12).

You can also use an `AUNode` instance as an element in a complex graph by defining the node to represent a complete audio processing subgraph. In this case, the I/O unit at the end of the subgraph must be a Generic Output unit—the one type of I/O unit that does not connect to device hardware.

In broad strokes, constructing an audio processing graph entails three tasks:

1. Adding nodes to a graph
2. Directly configuring the audio units represented by the nodes
3. Interconnecting the nodes

For details on these tasks and on the rest of the audio processing graph life cycle, refer to "Constructing Audio Unit Apps" (page 32). For a complete description of this rich API, see *Audio Unit Processing Graph Services Reference*.

An Audio Processing Graph Has Exactly One I/O Unit

Every audio processing graph has one I/O unit, whether you are doing recording, playback, or simultaneous I/O. The I/O unit can be any one of those available in iOS, depending on the needs of your app. For details on how I/O units fit into the architecture of an audio processing graph in various usage scenarios, see "Start by Choosing a Design Pattern" (page 32).

Graphs let you start and stop the flow of audio by way of the `AUGraphStart` and `AUGraphStop` functions. These functions, in turn, convey the start or stop message to the I/O unit by invoking its `AudioOutputUnitStart` or `AudioOutputUnitStop` function. In this way, a graph's I/O unit is in charge of the audio flow in the graph.

Audio Processing Graphs Provide Thread Safety

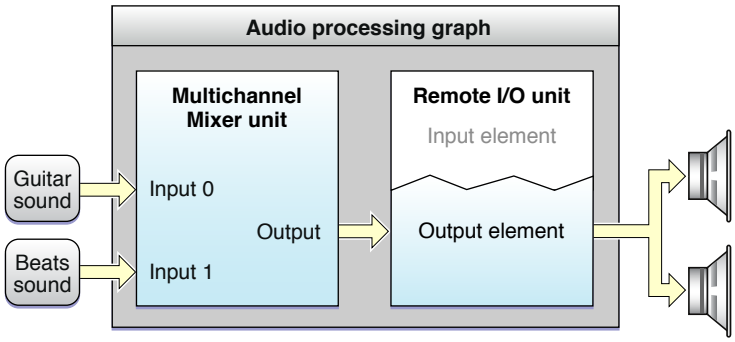
The audio processing graph API employs a "to-do list" metaphor to provide thread safety. Certain functions in this API add a unit of work to a list of changes to execute later. After you specify a complete set of changes, you then ask the graph to implement them.

Here are some common reconfigurations supported by the audio processing graph API, along with their associated functions:

- Adding or removing audio unit nodes (`AUGraphAddNode`, `AUGraphRemoveNode`)
- Adding or removing connections between nodes (`AUGraphConnectNodeInput`, `AUGraphDisconnectNodeInput`)
- Connecting a render callback function to an input bus of an audio unit (`AUGraphSetNodeInputCallback`)

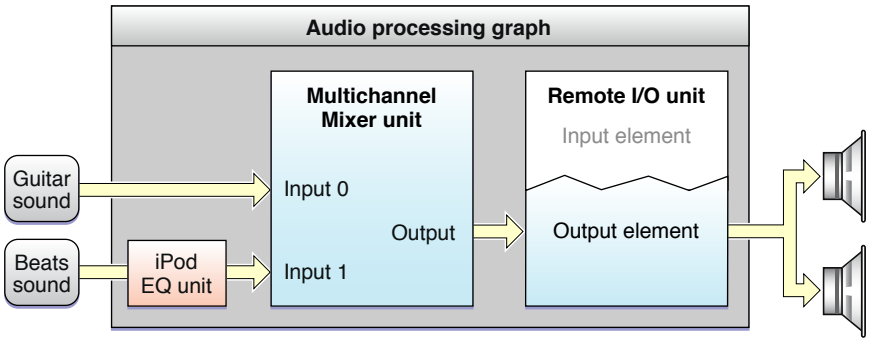
Let's look at an example of reconfiguring a running audio processing graph. Say, for example, you've built a graph that includes a Multichannel Mixer unit and a Remote I/O unit, for mixed playback of two synthesized sounds. You feed the sounds to two input buses of the mixer. The mixer output goes to the output element of the I/O unit and on to the output audio hardware. Figure 1-4 depicts this architecture.

Figure 1-4 A simple audio processing graph for playback



Now, say the user wants to insert an equalizer into one of the two audio streams. To do that, add an iPod EQ unit between the feed of one of the sounds and the mixer input that it goes to, as shown in Figure 1-5.

Figure 1-5 The same graph after inserting an equalizer



The steps to accomplish this live reconfiguration are as follows:

1. Disconnect the "beats sound" callback from input 1 of the mixer unit by calling `AUGraphDisconnectNodeInput`.
2. Add an audio unit node containing the iPod EQ unit to the graph. Do this by specifying the iPod EQ unit with an `AudioComponentDescription` structure, then calling `AUGraphAddNode`. At this point, the iPod EQ unit is instantiated but not initialized. It is owned by the graph but is not yet participating in the audio flow.
3. Configure and initialize the iPod EQ unit. In this example, this entails a few things:

- Call the `AudioUnitGetProperty` function to retrieve the stream format (`kAudioUnitProperty_StreamFormat`) from the mixer input.
- Call the `AudioUnitSetProperty` function twice, once to set that stream format on the iPod EQ unit's input and a second time to set it on the output. (For a complete description of how to configure an iPod EQ unit, see "Using Effect Units" (page 50).)
- Call the `AudioUnitInitialize` function to allocate resources for the iPod EQ unit and prepare it to process audio. This function call is not thread-safe, but you can (and must) perform it at this point in the sequence, when the iPod EQ unit is not yet participating actively in the audio processing graph because you have not yet called the `AUGraphUpdate` function.

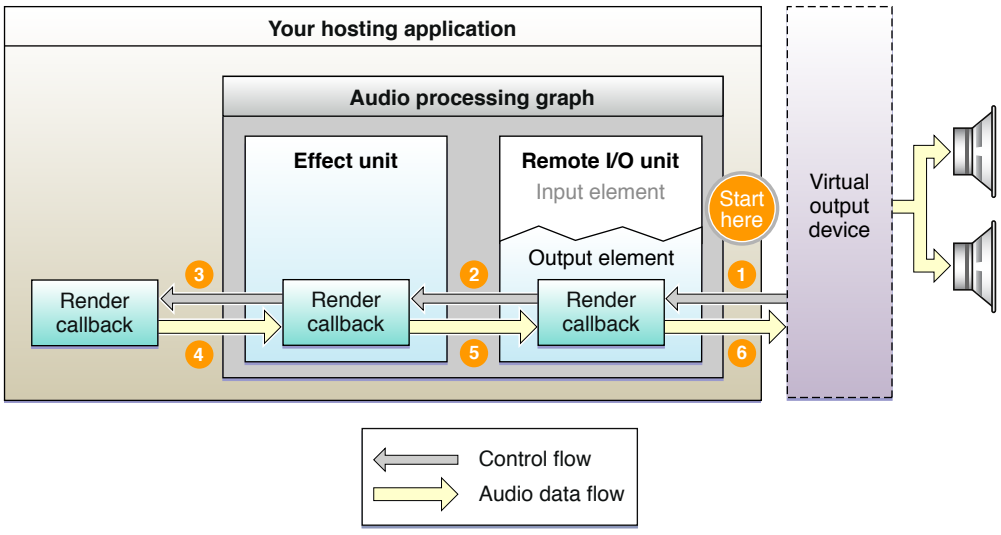
4. Attach the "beats sound" callback function to the input of the iPod EQ by calling `AUGraphSetNodeInputCallback`.

In the preceding list, steps 1, 2, and 4—all of them `AUGraph*` function calls—were added to the graph's "to-do" list. Call `AUGraphUpdate` to execute these pending tasks. On successful return of the `AUGraphUpdate` function, the graph has been dynamically reconfigured and the iPod EQ is in place and processing audio.

Audio Flows Through a Graph Using "Pull"

In an audio processing graph, the consumer calls the provider when it needs more audio data. There is a flow of requests for audio data, and this flow proceeds in a direction opposite to that of the flow of audio. Figure 1-6 illustrates this mechanism.

Figure 1-6 The pull mechanism of audio data flow



Each request for a set of data is known as a *render call* or, informally, as a *pull*. The figure represents render calls as gray "control flow" arrows. The data requested by a render call is more properly known as a set of *audio sample frames* (see "frame" in *Core Audio Glossary*).

In turn, a set of audio sample frames provided in response to a render call is known as a *slice*. (See "slice" in *Core Audio Glossary*.) The code that provides the slice is known as a *render callback function*, described in "Render Callback Functions Feed Audio to Audio Units" (page 24).

Here is how the pull proceeds in Figure 1-6:

1. After you call the `AUGraphStart` function, the virtual output device invokes the render callback of the Remote I/O unit's output element. This invocation asks for one slice of processed audio data frames.
2. The render callback function of the Remote I/O unit looks in its input buffers for audio data to process, to satisfy the render call. If there is data waiting to be processed, the Remote I/O unit uses it. Otherwise, and as shown in the figure, it instead invokes the render callback of whatever your app has connected to its input. In this example, the Remote I/O unit's input is connected to an effect unit's output. So, the I/O unit pulls on the effect unit, asking for a slice of audio frames.
3. The effect unit behaves just as the Remote I/O unit did. When it needs audio data, it gets it from its input connection. In this example, the effect unit pulls on your app's render callback function.
4. Your app's render callback function is the final recipient of the pull. It supplies the requested frames to the effect unit.
5. The effect unit processes the slice supplied by your app's render callback. The effect unit then supplies the processed data that were previously requested (in step 2) to the Remote I/O unit.
6. The Remote I/O unit processes the slice provided by the effect unit. The Remote I/O unit then supplies the processed slice originally requested (in step 1) to the virtual output device. This completes one cycle of pull.

Render Callback Functions Feed Audio to Audio Units

To provide audio from disk or memory to an audio unit input bus, convey it using a render callback function that conforms to the `AURenderCallback` prototype. The audio unit input invokes your callback when it needs another slice of sample frames, as described in "Audio Flows Through a Graph Using "Pull"" (page 23).

The process of writing a render callback function is perhaps the most creative aspect of designing and building an audio unit application. It's your opportunity to generate or alter sound in any way you can imagine and code.

At the same time, render callbacks have a strict performance requirement that you must adhere to. A render callback lives on a real-time priority thread on which subsequent render calls arrive asynchronously. The work you do in the body of a render callback takes place in this time-constrained environment. If your callback is still producing sample frames in response to the previous render call when the next render call arrives, you get a gap in the sound. For this reason you must not take locks, allocate memory, access the file system or a network connection, or otherwise perform time-consuming tasks in the body of a render callback function.

Understanding the Audio Unit Render Callback Function

Listing 1-5 shows the header of a render callback function that conforms to the `AURenderCallback` prototype. This section describes the purpose of each of its parameters in turn and explains how to use each one.

Listing 1-5 A render callback function header

```
static OSStatus MyAURenderCallback (
    void                        *inRefCon,
    AudioUnitRenderActionFlags  *ioActionFlags,
    const AudioTimeStamp        *inTimeStamp,
    UInt32                      inBusNumber,
    UInt32                      inNumberFrames,
    AudioBufferList             *ioData
) { /* callback body */ }
```

The `inRefCon` parameter points to a programmatic context you specify when attaching the callback to an audio unit input (see "Write and Attach Render Callback Functions" (page 41)). The purpose of this context is to provide the callback function with any audio input data or state information it needs to calculate the output audio for a given render call.

The `ioActionFlags` parameter lets a callback provide a hint to the audio unit that there is no audio to process. Do this, for example, if your app is a synthetic guitar and the user is not currently playing a note. During a callback invocation for which you want to output silence, use a statement like the following in the body of the callback:

```
*ioActionFlags |= kAudioUnitRenderAction_OutputIsSilence;
```

When you want to produce silence, you must also explicitly set the buffers pointed at by the `ioData` parameter to 0. There's more about this in the description for that parameter.

The `inTimeStamp` parameter represents the time at which the callback was invoked. It contains an `AudioTimeStamp` structure, whose `mSampleTime` field is a sample-frame counter. On each invocation of the callback, the value of the `mSampleTime` field increments by the number in the `inNumberFrames` parameter. If your app is a sequencer or a drum machine, for example, you can use the `mSampleTime` value for scheduling sounds.

The `inBusNumber` parameter indicates the audio unit bus that invoked the callback, allowing you to branch within the callback depending on this value. In addition, when attaching a callback to an audio unit, you can specify a different context (`inRefCon`) for each bus.

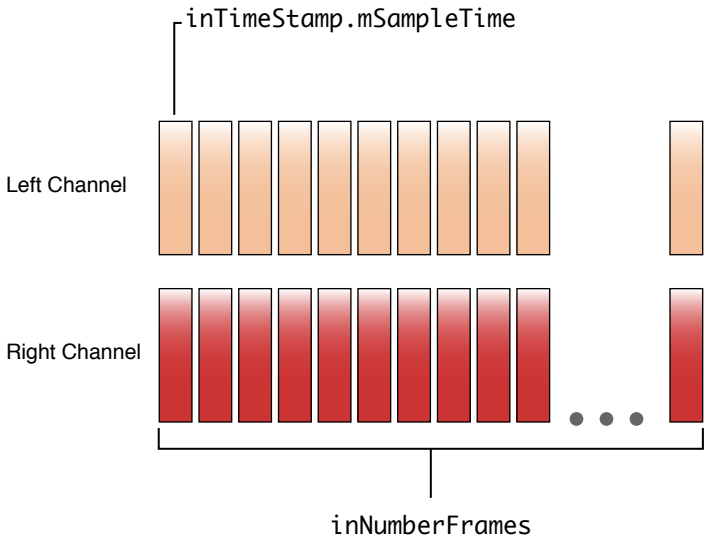
The `inNumberFrames` parameter indicates the number of audio sample frames that the callback is being asked to provide on the current invocation. You provide those frames to the buffers in the `ioData` parameter.

The `ioData` parameter points to the audio data buffers that the callback must fill when it is invoked. The audio you place into these buffers must conform to the audio stream format of the bus that invoked the callback.

If you are playing silence for a particular invocation of the callback, explicitly set these buffers to 0, such as by using the `memset` function.

Figure 1-7 depicts a pair of noninterleaved stereo buffers in an `ioData` parameter. Use the elements of the figure to visualize the details of `ioData` buffers that your callback needs to fill.

Figure 1-7 The `ioData` buffers for a stereo render callback function



Audio Stream Formats Enable Data Flow

When working with audio data at the individual sample level, as you are when using audio units, it's not enough to specify the correct data type to represent the audio. The layout of the bits in a single audio sample value has meaning, so a data type like `Float32` or `UInt16` is not expressive enough. In this section you learn about Core Audio's solution to this problem.

Working with the AudioStreamBasicDescription structure

The currency for moving audio values around in your app, and between your app and audio hardware, is the `AudioStreamBasicDescription` structure, shown in Listing 1-6 and described fully in *Core Audio Data Types Reference*.

Listing 1-6 The `AudioStreamBasicDescription` structure

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

Because the name `AudioStreamBasicDescription` is long, it's often abbreviated in conversation and documentation as *ASBD*. To define values for the fields of an ASBD, write code similar to that shown in Listing 1-7.

Listing 1-7 Defining an ASBD for a stereo stream

```
size_t bytesPerSample = sizeof (AudioUnitSampleType);
AudioStreamBasicDescription stereoStreamFormat = {0};

stereoStreamFormat.mFormatID          = kAudioFormatLinearPCM;
stereoStreamFormat.mFormatFlags       = kAudioFormatFlagsAudioUnitCanonical;
```

```
stereoStreamFormat.mBytesPerPacket    = bytesPerSample;
stereoStreamFormat.mBytesPerFrame     = bytesPerSample;
stereoStreamFormat.mFramesPerPacket   = 1;
stereoStreamFormat.mBitsPerChannel    = 8 * bytesPerSample;
stereoStreamFormat.mChannelsPerFrame  = 2;          // 2 indicates stereo
stereoStreamFormat.mSampleRate        = graphSampleRate;
```

To start, determine the data type to represent one audio sample value. This example uses the `AudioUnitSampleType` defined type, the recommended data type for most audio units. In iOS, `AudioUnitSampleType` is defined to be an 8.24 fixed-point integer. The first line in Listing 1-7 calculates the number of bytes in the type; that number is required when defining some of the field values of an ASBD, as you can see in the listing.

Next, still referring to Listing 1-7, declare a variable of type `AudioStreamBasicDescription` and initialize its fields to 0 to ensure that no fields contain garbage data. Do not skip this zeroing step; if you do, you are certain to run into trouble later.

Now define the ASBD field values. Specify `kAudioFormatLinearPCM` for the `mFormatID` field. Audio units use uncompressed audio data, so this is the correct format identifier to use whenever you work with audio units.

Next, for most audio units, specify the `kAudioFormatFlagsAudioUnitCanonical` metaflag for the `mFormatFlags` field. This flag is defined in `CoreAudio.framework/CoreAudioTypes.h` as follows:

```
kAudioFormatFlagsAudioUnitCanonical = kAudioFormatFlagIsFloat |
                                  kAudioFormatFlagsNativeEndian |
                                       kAudioFormatFlagIsPacked |
                                kAudioFormatFlagIsNonInterleaved
```

This metaflag takes care of specifying all of the layout details for the bits in a linear PCM sample value of type `AudioUnitSampleType`.

Certain audio units employ an atypical audio data format, requiring a different data type for samples and a different set of flags for the `mFormatFlags` field. For example, the 3D Mixer unit requires the `UInt16` data type for its audio sample values and requires the ASBD's `mFormatFlags` field to be set to `kAudioFormatFlagsCanonical`. When working with a particular audio unit, be careful to use the correct data format and the correct format flags. (See "Using Specific Audio Units" (page 46).)

Continuing through Listing 1-7 (page 27), the next four fields further specify the organization and meaning of the bits in a sample frame. Set these fields—`mBytesPerPacket`, `mBytesPerFrame`, `mFramesPerPacket`, and `mBitsPerChannel` fields—according to the nature of the audio stream you are using. To learn the meaning of each of these fields, refer to the documentation for the `AudioStreamBasicDescription` structure. You can see an example of filled-out ASBDs in the sample code project *Audio Mixer (MixerHost)* .

Set the ASBD's `mChannelsPerFrame` field according to the number of channels in the stream—1 for mono audio, 2 for stereo, and so on.

Finally, set the `mSampleRate` field according to the sample rate that you are using throughout your app. "Understanding Where and How to Set Stream Formats" (page 29) explains the importance of avoiding sample rate conversions. "Configure Your Audio Session" (page 38) explains how to ensure that your application's sample rate matches the audio hardware sample rate.

Rather than specify an ASBD field by field as you've seen here, you can use the C++ utility methods provided in the `CAStreamBasicDescription.h` file (`/Developer/Extras/CoreAudio/PublicUtility/`). In particular, view the `SetAUCanonical` and `SetCanonical` C++ methods. These specify the correct way to derive ASBD field values given three factors:

- Whether the stream is for I/O (`SetCanonical`) or for audio processing (`SetAUCanonical`)
- How many channels you want the stream format to represent
- Whether you want the stream format interleaved or noninterleaved

Whether or not you include the `CAStreamBasicDescription.h` file in your project to use its methods directly, Apple recommends that you study that file to learn the correct way to work with an `AudioStreamBasicDescription` structure.

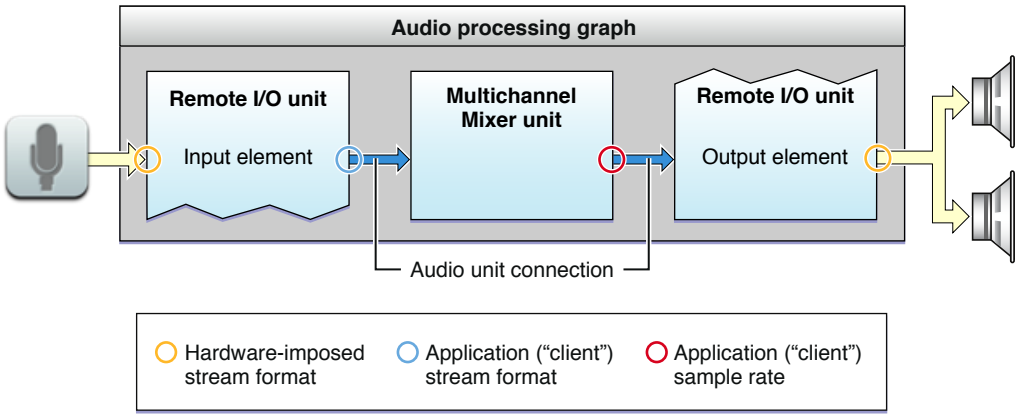
See "Troubleshooting Tips" (page 44) for ideas on how to fix problems related to audio data stream formats.

Understanding Where and How to Set Stream Formats

You must set the audio data stream format at critical points in an audio processing graph. At other points, the system sets the format. At still other points, audio unit connections propagate a stream format from one audio unit to another.

The audio input and output hardware on an iOS device have system-determined audio stream formats. These formats are always uncompressed, in linear PCM format, and interleaved. The system imposes these formats on the outward-facing sides of the I/O unit in an audio processing graph, as depicted in Figure 1-8.

Figure 1-8 Where to set audio data stream formats



In the figure, the microphone represents the input audio hardware. The system determines the input hardware's audio stream format and imposes it onto the input scope of the Remote I/O unit's input element.

Similarly, the loudspeakers in the figure represent the output audio hardware. The system determines the output hardware's stream format and imposes it onto the output scope of the Remote I/O unit's output element.

Your application is responsible for establishing the audio stream formats on the inward-facing sides of the I/O unit's elements. The I/O unit performs any necessary conversion between your application formats and the hardware formats. Your application is also responsible for setting stream formats wherever else they are required in a graph. In some cases, such as at the output of the Multichannel Mixer unit in Figure 1-8, you need to set only a portion of the format—specifically, the sample rate. "Start by Choosing a Design Pattern" (page 32) shows you where to set stream formats for various types of audio unit apps. "Using Specific Audio Units" (page 46) lists the stream format requirements for each iOS audio unit.

A key feature of an audio unit connection, as shown in Figure 1-8 (page 30), is that the connection propagates the audio data stream format from the output of its source audio unit to the input of its destination audio unit. This is a critical point so it bears emphasizing: Stream format propagation takes place by way of an audio unit connection and in one direction only—from the output of a source audio unit to an input of a destination audio unit.

Take advantage of format propagation. It can significantly reduce the amount of code you need to write. For example, when connecting the output of a Multichannel Mixer unit to the Remote I/O unit for playback, you do not need to set the stream format for the I/O unit. It is set appropriately by the connection between the audio units, based on the output stream format of the mixer (see Figure 1-8 (page 30)).

Stream format propagation takes place at one particular point in an audio processing graph's life cycle—namely, upon initialization. See "Initialize and Start the Audio Processing Graph" (page 44).

You have great flexibility in defining your application audio stream formats. However, whenever possible, use the sample rate that the hardware is using. When you do, the I/O unit need not perform sample rate conversion. This minimizes energy usage—an important consideration in a mobile device—and maximizes audio quality. To learn about working with the hardware sample rate, see "Configure Your Audio Session" (page 38).

Constructing Audio Unit Apps

Now that you understand how audio unit hosting works, as explained in "Audio Unit Hosting Fundamentals" (page 10), you are well prepared to build the audio unit portion of your app. The main steps are choosing a design pattern and then writing the code to implement that pattern.

Start by Choosing a Design Pattern

There are a half dozen basic design patterns for hosting audio units in an iOS app. Begin by picking the one that most closely represents what you want your app to do with audio. As you learn each pattern, notice the common features. Every pattern:

- Has exactly one I/O unit.
- Uses a single audio stream format throughout the audio processing graph—although there can be variations on that format, such as mono and stereo streams feeding a mixer unit.
- Requires that you set the stream format, or portions of the stream format, at specific locations.

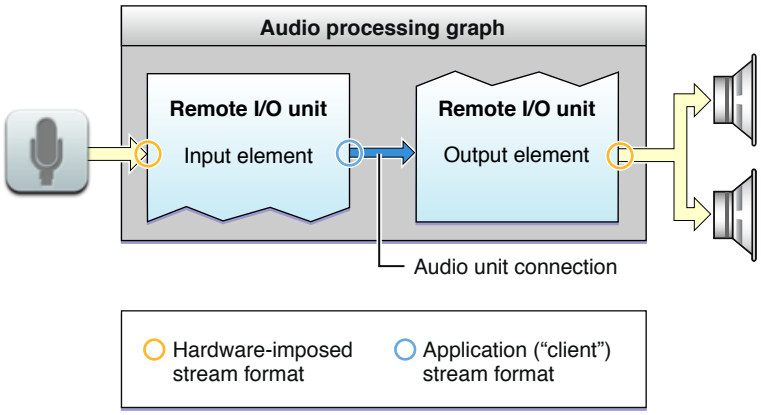
Setting stream formats correctly is essential to establishing audio data flow. Most of these patterns rely on automatic propagation of audio stream formats from source to destination, as provided by an audio unit connection. Take advantage of this propagation when you can because it reduces the amount of code to write and maintain. At the same time, be sure that you understand where it is required for you to set stream formats. For example, you must set the full stream format on the input *and* output of an iPod EQ unit. Refer to the usage tables in "Using Specific Audio Units" (page 46) for all iOS audio unit stream format requirements.

In most cases, the design patterns in this chapter employ an audio processing graph (of type `AUGraph`). You could implement any one of these patterns without using a graph, but using one simplifies the code and supports dynamic reconfiguration, as described in "Audio Processing Graphs Manage Audio Units" (page 20).

I/O Pass Through

The I/O pass-through pattern sends incoming audio directly to the output hardware, with no option to work with the audio data. Although this isn't of much practical value, building an audio unit hosting app based on this pattern is a good way to verify and cement your understanding of audio unit concepts. Figure 2-1 illustrates this pattern.

Figure 2-1 Simultaneous I/O pass through



As you can see in the figure, the audio input hardware imposes its stream format on the outward-facing side of the Remote I/O unit's input element. You, in turn, specify the format that you want to use on the inward-facing side of this element. The audio unit performs format conversion as needed. To avoid unnecessary sample rate conversion, be sure to use the audio hardware sample rate when defining your stream format.

The input element is disabled by default, so be sure to enable it; otherwise, audio cannot flow.

The pattern shown in Figure 2-1 takes advantage of the audio unit connection between the two Remote I/O elements. Specifically, you do not set a stream format on the input scope of the audio unit's output element. The connection propagates the format you specified for the input element.

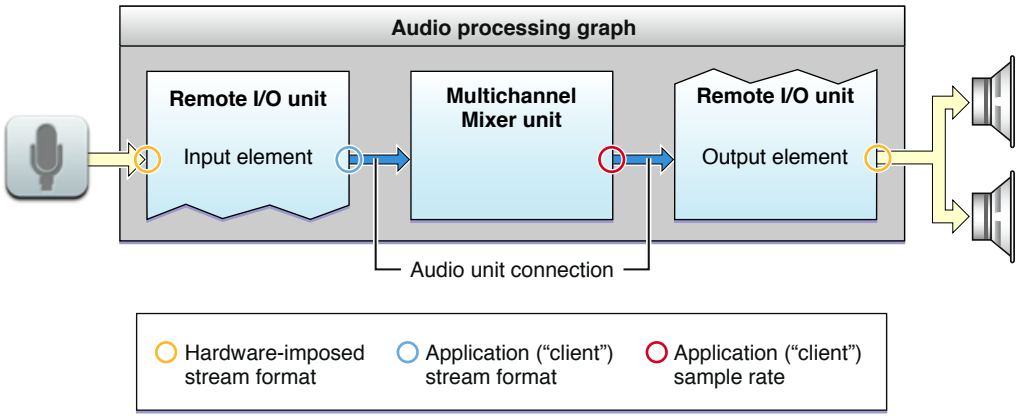
The outward-facing side of the output element takes on the audio output hardware's stream format, and the output element performs format conversion for the outgoing audio as needed.

Using this pattern, you need not configure any audio data buffers.

I/O Without a Render Callback Function

Adding one or more other audio units between the Remote I/O unit's elements lets you construct a more interesting app. For example, you could use a Multichannel Mixer unit to position the incoming microphone audio in a stereo field or to provide output volume control. In this design pattern, there is still no callback function in play, as shown in Figure 2-2. This simplifies the pattern but limits its utility. Without a render callback function you don't have a means to manipulate the audio directly.

Figure 2-2 Simultaneous I/O without a render callback function



In this pattern, you configure both elements of the Remote I/O unit just as you do in the pass-through pattern. To set up the Multichannel Mixer unit, you must set the sample rate of your stream format on the mixer output, as indicated in Figure 2-2.

The mixer's input stream format is established automatically by propagation from the output of the Remote I/O unit's input element, by way of the audio unit connection. Similarly, the stream format for the input scope of the Remote I/O unit's output element is established by the audio unit connection, thanks to propagation from the mixer unit output.

In any instance of this pattern—indeed, whenever you use other audio units in addition to an I/O unit—you must set the `kAudioUnitProperty_MaximumFramesPerSlice` property as described in *Audio Unit Properties Reference*.

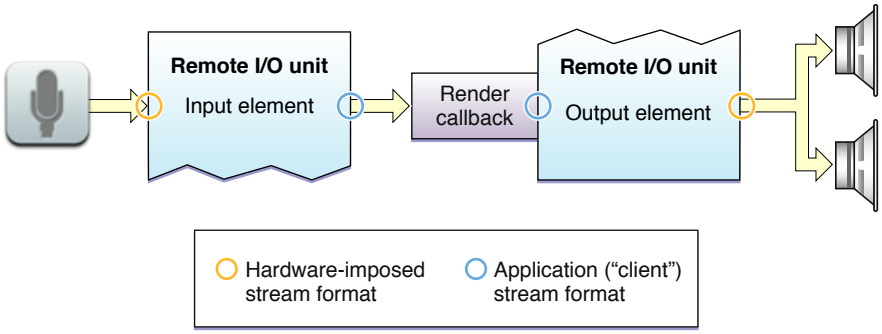
As with the pass-through pattern, you need not configure any audio data buffers.

I/O with a Render Callback Function

By placing a render callback function between the input and output elements of a Remote I/O unit, you can manipulate incoming audio before it reaches the output hardware. In a very simple case, you could use the render callback function to adjust output volume. However, you could add tremolo, ring-modulation, echo, or

other effects. By making use of the Fourier transforms and convolution functions available in the Accelerate framework (see *Accelerate Framework Reference*), your possibilities are endless. This pattern is depicted in Figure 2-3.

Figure 2-3 Simultaneous I/O with a render callback function



As you can see in the figure, this pattern uses both elements of the Remote I/O unit, as in the previous patterns in this chapter. Attach your render callback function to the input scope of the output element. When that element needs another set of audio sample values, it invokes your callback. Your callback, in turn, obtains fresh samples by invoking the render callback function of the Remote I/O unit's input element.

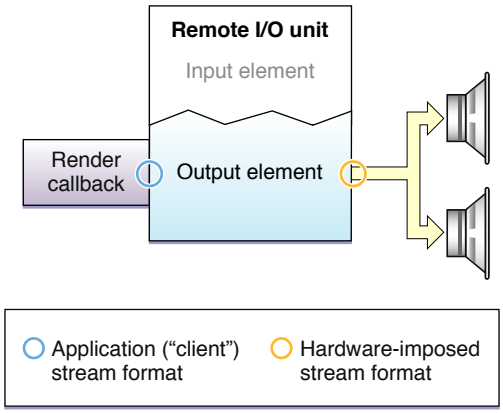
Just as for the other I/O patterns, you must explicitly enable input on the Remote I/O unit, because input is disabled by default. And, as for the other I/O patterns, you need not configure any audio data buffers.

Notice that when you establish an audio path from one audio unit to another using a render callback function, as in this pattern, the callback takes the place of an audio unit connection.

Output-Only with a Render Callback Function

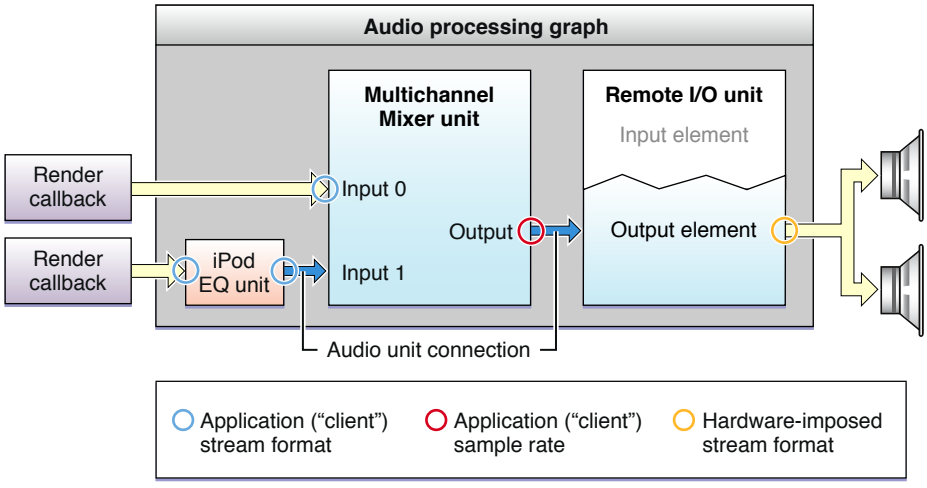
Choose this pattern for musical games and synthesizers—apps for which you are generating sounds and need maximum responsiveness. At its simplest, this pattern involves one render callback function connected directly to the input scope of a Remote I/O unit's output element, as shown in Figure 2-4.

Figure 2-4 Output-only with a render callback function



You can use this same pattern to build an app with a more complex audio structure. For example, you might want to generate several sounds, mix them together, and then play them through the device's output hardware. Figure 2-5 shows such a case. Here, the pattern employs an audio processing graph and two additional audio units, a Multichannel Mixer and an iPod EQ.

Figure 2-5 A more complex example of output-only with a render callback function



In the figure, notice that the iPod EQ requires you to set your full stream format on both input and output. The Multichannel Mixer, on the other hand, needs only the correct sample rate to be set on its output. The full stream format is then propagated by the audio unit connection from the mixer's output to the input scope of the Remote I/O unit's output element. These usage details, and other specifics of using the various iOS audio units, are described in "Using Specific Audio Units" (page 46).

For each of the Multichannel Mixer unit inputs, as you see in Figure 2-5, the full stream format is set. For input 0, you set it explicitly. For input 1, the format is propagated by the audio unit connection from the output of the iPod EQ unit. In general, you must account for the stream-format needs of each audio unit individually.

Other Audio Unit Hosting Design Patterns

There are two other main design patterns for audio units hosting. To record or analyze audio, create an input-only app with a render callback function. The callback function is invoked by your application, and it in turn invokes the render method of the Remote I/O unit's input element. However, in most cases, a better choice for an app like this is to use an input audio queue object (of type `AudioQueueRef` instantiated using the `AudioQueueNewInput` function), as explained in *Audio Queue Services Programming Guide*. Using an audio queue object provides a great deal more flexibility because its render callback function is not on a realtime thread.

To perform offline audio processing, use a Generic Output unit. Unlike the Remote I/O unit, this audio unit does not connect to the device's audio hardware. When you use it to send audio to your application, it depends on your application to invoke its render method.

Constructing Your App

No matter which design pattern you choose, the steps for constructing an audio unit hosting app are basically the same:

1. Configure your audio session.
2. Specify audio units.
3. Create an audio processing graph, then obtain the audio units.
4. Configure the audio units.
5. Connect the audio unit nodes.
6. Provide a user interface.
7. Initialize and then start the audio processing graph.

Configure Your Audio Session

The first step in building an audio unit application is the same step as for any iOS audio application: You configure the audio session. The characteristics of the audio session largely determine your app's audio capabilities as well as its interactivity with the rest of the system. Start by specifying the sample rate you want to use in your application, as shown here:

```
self.graphSampleRate = 44100.0; // Hertz
```

Next, employ the audio session object to request that the system use your preferred sample rate as the device hardware sample rate, as shown in Listing 2-1. The intent here is to avoid sample rate conversion between the hardware and your app. This maximizes CPU performance and sound quality, and minimizes battery drain.

Listing 2-1 Configuring an audio session

```
NSError *audioSessionError = nil;
AVAudioSession *mySession = [AVAudioSession sharedInstance];     // 1
[mySession setPreferredHardwareSampleRate: graphSampleRate       // 2
                                    error: &audioSessionError];
[mySession setCategory: AVAudioSessionCategoryPlayAndRecord      // 3
                                    error: &audioSessionError];
[mySession setActive: YES                                        // 4
               error: &audioSessionError];
self.graphSampleRate = [mySession currentHardwareSampleRate];    // 5
```

The preceding lines do the following:

1. Obtain a reference to the singleton audio session object for your application.
2. Request a hardware sample rate. The system may or may not be able to grant the request, depending on other audio activity on the device.
3. Request the audio session category you want. The "play and record" category, specified here, supports audio input and output.
4. Request activation of your audio session.
5. After audio session activation, update your own sample rate variable according to the actual sample rate provided by the system.

There's one other hardware characteristic you may want to configure: audio hardware I/O buffer duration. The default duration is about 23 ms at a 44.1 kHz sample rate, equivalent to a slice size of 1,024 samples. If I/O latency is critical in your app, you can request a smaller duration, down to about 0.005 ms (equivalent to 256 samples), as shown here:

```
self.ioBufferDuration = 0.005;
[mySession setPreferredIOBufferDuration: ioBufferDuration
                                 error: &audioSessionError];
```

For a complete explanation of how to configure and use the audio session object, see *Audio Session Programming Guide*.

Specify the Audio Units You Want

At runtime, after your audio session configuration code has run, your app has not yet acquired audio units. You specify each one that you want by using an `AudioComponentDescription` structure. See "Use Identifiers to Specify and Obtain Audio Units" (page 13) for how to do this. The identifier keys for each iOS audio unit are listed in "Identifier Keys for Audio Units" (page 52).

Audio unit specifiers in hand, you then build an audio processing graph according to the pattern you've picked.

Build an Audio Processing Graph

In this step, you create the skeleton of one of the design patterns explained in the first part of this chapter. Specifically, you:

1. Instantiate an `AUGraph` opaque type. The instance represents the audio processing graph.
2. Instantiate one or more `AUNode` opaque types, each of which represents one audio unit in the graph.
3. Add the nodes to the graph.
4. Open the graph and instantiate the audio units.
5. Obtain references to the audio units.

Listing 2-2 shows how to perform these steps for a graph that contains a Remote I/O unit and a Multichannel Mixer unit. It assumes you've already defined an `AudioComponentDescription` structure for each of these audio units.

Listing 2-2 Building an audio processing graph

```
AUGraph processingGraph;
```

```
NewAUGraph (&processingGraph);

AUNode ioNode;
AUNode mixerNode;

AUGraphAddNode (processingGraph, &ioUnitDesc, &ioNode);
AUGraphAddNode (processingGraph, &mixerDesc, &mixerNode);
```

The `AUGraphAddNode` function calls make use of the audio unit specifiers `ioUnitDesc` and `mixerDesc`. At this point, the graph is instantiated and owns the nodes that you'll use in your app. To open the graph and instantiate the audio units, call `AUGraphOpen`:

```
AUGraphOpen (processingGraph);
```

Then, obtain references to the audio unit instances by way of the `AUGraphNodeInfo` function, as shown here:

```
AudioUnit ioUnit;
AudioUnit mixerUnit;

AUGraphNodeInfo (processingGraph, ioNode, NULL, &ioUnit);
AUGraphNodeInfo (processingGraph, mixerNode, NULL, &mixerUnit);
```

The `ioUnit` and `mixerUnit` variables now hold references to the audio unit instances in the graph, allowing you to configure and then interconnect the audio units.

Configure the Audio Units

Each iOS audio unit requires its own configuration, as described in "Using Specific Audio Units" (page 46). However, some configurations are common enough that all iOS audio developers should be familiar with them.

The Remote I/O unit, by default, has output enabled and input disabled. If your app performs simultaneous I/O, or uses input only, you must reconfigure the I/O unit accordingly. For details, see the `kAudioOutputUnitProperty_EnableIO` property in *Audio Unit Properties Reference*.

All iOS audio units, with the exception of the Remote I/O and Voice-Processing I/O units, need their `kAudioUnitProperty_MaximumFramesPerSlice` property configured. This property ensures that the audio unit is prepared to produce a sufficient number of frames of audio data in response to a render call. For details, see `kAudioUnitProperty_MaximumFramesPerSlice` in *Audio Unit Properties Reference*.

All audio units need their audio stream format defined on input, output, or both. For an explanation of audio stream formats, see "Audio Stream Formats Enable Data Flow" (page 27). For the specific stream format requirements of the various iOS audio units, see "Using Specific Audio Units" (page 46).

Write and Attach Render Callback Functions

For design patterns that employ render callback functions, you must write those functions and then attach them at the correct points. "Render Callback Functions Feed Audio to Audio Units" (page 24) describes what these callbacks do and explains how they work. For examples of working callbacks, view the various audio unit sample code projects in the iOS Reference Library including *Audio Mixer (MixerHost)* and *aurioTouch*, and *SynthHost*.

When audio is not flowing, you can attach a render callback immediately by using the audio unit API, as shown in Listing 2-3.

Listing 2-3 Attaching a render callback immediately

```
AURenderCallbackStruct callbackStruct;
callbackStruct.inputProc        = &renderCallback;
callbackStruct.inputProcRefCon  = soundStructArray;

AudioUnitSetProperty (
    myIOUnit,
    kAudioUnitProperty_SetRenderCallback,
    kAudioUnitScope_Input,
    0,                  // output element
    &callbackStruct,
    sizeof (callbackStruct)
);
```

You can attach a render callback in a thread-safe manner, even when audio is flowing, by using the audio processing graph API. Listing 2-4 shows how.

Listing 2-4 Attaching a render callback in a thread-safe manner

```
AURenderCallbackStruct callbackStruct;
callbackStruct.inputProc        = &renderCallback;
callbackStruct.inputProcRefCon  = soundStructArray;

AUGraphSetNodeInputCallback (
    processingGraph,
    myIONode,
    0,                  // output element
    &callbackStruct
);
// ... some time later
Boolean graphUpdated;
AUGraphUpdate (processingGraph, &graphUpdated);
```

Connect the Audio Unit Nodes

In most cases, it's best—and easier—to establish or break connections between audio units using the `AUGraphConnectNodeInput` and `AUGraphDisconnectNodeInput` functions in the audio processing graph API. These functions are thread-safe and avoid the coding overhead of defining connections explicitly, as you must do when not using a graph.

Listing 2-5 shows how to connect the output of a mixer node to the input of an I/O unit output element using the audio processing graph API.

Listing 2-5 Connecting two audio unit nodes using the audio processing graph API

```
AudioUnitElement mixerUnitOutputBus  = 0;
AudioUnitElement ioUnitOutputElement = 0;


AUGraphConnectNodeInput (
    processingGraph,
    mixerNode,           // source node
    mixerUnitOutputBus,  // source node bus
    iONode,              // destination node
    ioUnitOutputElement  // desinatation node element
);
```

You can, alternatively, establish and break connections between audio units directly by using the audio unit property mechanism. To do so, use the `AudioUnitSetProperty` function along with the `kAudioUnitProperty_MakeConnection` property, as shown in Listing 2-6. This approach requires that you define an `AudioUnitConnection` structure for each connection to serve as its property value.

Listing 2-6 Connecting two audio units directly

```
AudioUnitElement mixerUnitOutputBus  = 0;
AudioUnitElement ioUnitOutputElement = 0;

AudioUnitConnection mixerOutToIoUnitIn;
mixerOutToIoUnitIn.sourceAudioUnit    = mixerUnitInstance;
mixerOutToIoUnitIn.sourceOutputNumber = mixerUnitOutputBus;
mixerOutToIoUnitIn.destInputNumber    = ioUnitOutputElement;

AudioUnitSetProperty (
    ioUnitInstance,                   // connection destination
    kAudioUnitProperty_MakeConnection,  // property key
    kAudioUnitScope_Input,              // destination scope
    ioUnitOutputElement,                // destination element
    &mixerOutToIoUnitIn,                // connection definition
    sizeof (mixerOutToIoUnitIn)
);
```

Provide a User Interface

At this point in constructing your app, the audio units—and, typically, the audio processing graph—are fully built and configured. In many cases, you'll then want to provide a user interface to let your users fine-tune the audio behavior. You tailor the user interface to allow the user to adjust specific audio unit parameters and, in some unusual cases, audio unit properties. In either case, the user interface should also provide visual feedback regarding the current settings.

"Use Parameters and UIKit to Give Users Control" (page 18) explains the basics of constructing a user interface to let a user control a parameter value. For a working example, view the sample code project *Audio Mixer (MixerHost)*.

The iPod EQ unit is one of the unusual cases in that, to change its active equalization curve, you change the value of the `kAudioUnitProperty_PresentPreset` property. You can do this whether or not audio is running. For a working example, view the sample code project *iPhoneMixerEQGraphTest*.

Initialize and Start the Audio Processing Graph

Before you can start audio flow, an audio processing graph must be initialized by calling the `AUGraphInitialize` function. This critical step:

- Initializes the audio units owned by the graph by automatically invoking the `AudioUnitInitialize` function individually for each one. (If you were to construct a processing chain without using a graph, you would have to explicitly initialize each audio unit in turn.)
- Validates the graph's connections and audio data stream formats.
- Propagates stream formats across audio unit connections.

Listing 2-7 shows how to use `AUGraphInitialize`.

Listing 2-7 Initializing and starting an audio processing graph

```
OSStatus result = AUGraphInitialize (processingGraph);
// Check for error. On successful initialization, start the graph...
AUGraphStart (processingGraph);

// Some time later
AUGraphStop (processingGraph);
```

Troubleshooting Tips

Whenever a Core Audio function provides a return value, capture that value and check for success or failure. On failure, make use of Xcode's debugging features as described in *Xcode Debugging Guide*. If using an Objective-C method in your app, such as for configuring your audio session, take advantage the `error` parameter in the same way.

Be aware of dependencies between function calls. For example, you can start an audio processing graph only after you successfully initialize it. Check the return value of `AUGraphInitialize`. If the function returns successfully, you can start the graph. If it fails, determine what went wrong. Check that all of your audio unit function calls leading up to initialization returned successfully. For an example of how to do this, look at the `-configureAndInitializeAudioProcessingGraph` method in the sample code project *Audio Mixer (MixerHost)*.

Second, if graph initialization is failing, take advantage of the `CAShow` function. This function prints out the state of the graph to the Xcode console. The sample code project *Audio Mixer (MixerHost)* demonstrates this technique as well.

Ensure that you are initializing each of your `AudioStreamBasicDescription` structures to 0, as follows:

```
AudioStreamBasicDescription stereoStreamFormat = {0};
```

Initializing the fields of an ASBD to 0 ensures that no fields contain garbage data. (In the case of declaring a data structure in external storage—for example, as an instance variable in a class declaration—its fields are automatically initialized to 0 and you need not initialize them yourself.)

To print out the field values of an `AudioStreamBasicDescription` structure to the Xcode console, which can be very useful during development, use code like that shown in Listing 2-8.

Listing 2-8 A utility method to print field values for an `AudioStreamBasicDescription` structure

```
- (void) printASBD: (AudioStreamBasicDescription) asbd {

    char formatIDString[5];
    UInt32 formatID = CFSwapInt32HostToBig (asbd.mFormatID);
    bcopy (&formatID, formatIDString, 4);
    formatIDString[4] = '\0';

    NSLog (@"  Sample Rate:         %10.0f",  asbd.mSampleRate);
    NSLog (@"  Format ID:           %10s",    formatIDString);
    NSLog (@"  Format Flags:        %10X",    asbd.mFormatFlags);
    NSLog (@"  Bytes per Packet:    %10d",    asbd.mBytesPerPacket);
    NSLog (@"  Frames per Packet:   %10d",    asbd.mFramesPerPacket);
    NSLog (@"  Bytes per Frame:     %10d",    asbd.mBytesPerFrame);
    NSLog (@"  Channels per Frame:  %10d",    asbd.mChannelsPerFrame);
    NSLog (@"  Bits per Channel:    %10d",    asbd.mBitsPerChannel);
}
```

This utility method can quickly reveal problems in an ASBD.

When defining an ASBD for an audio unit stream format, take care to ensure you are following the "Recommended stream format attributes" and "Stream format notes" in the usage tables in "Using Specific Audio Units" (page 46). Do not deviate from those recommendations unless you have a specific reason to.

Using Specific Audio Units

Each iOS audio unit has certain things in common with all others and certain things unique to itself. Earlier chapters in this document described the common aspects, among them the need to find the audio unit at runtime, instantiate it, and ensure that its stream formats are set appropriately. This chapter explains the differences among the audio units and provides specifics on how to use them.

Later in the chapter, "Identifier Keys for Audio Units" (page 52) lists the codes you need to locate the dynamically-linkable libraries for each audio unit at runtime.

Using I/O Units

iOS provides three I/O (input/output) units. The vast majority of audio-unit applications use the Remote I/O unit, which connects to input and output audio hardware and provides low-latency access to individual incoming and outgoing audio sample values. For VoIP apps, the Voice-Processing I/O unit extends the Remote I/O unit by adding acoustic echo cancelation and other features. To send audio back to your application rather than to output audio hardware, use the Generic Output unit.

Remote I/O Unit

The Remote I/O unit (subtype `kAudioUnitSubType_RemoteIO`) connects to device hardware for input, output, or simultaneous input and output. Use it for playback, recording, or low-latency simultaneous input and output where echo cancelation is not needed.

The device's audio hardware imposes its audio stream formats on the outward-facing sides of the Remote I/O unit, as described in "Understanding Where and How to Set Stream Formats" (page 29). The audio unit provides format conversion between the hardware audio formats and your application audio format, doing so by way of an included Format Converter audio unit.

For sample code that shows how to use this audio unit, see the sample code project *aurioTouch*.

Table 3-1 provides usage details for this audio unit.

Table 3-1 Using the Remote I/O unit

Audio unit feature	Details
Elements	One input element: element 1. One output element: element 0. By default, the input element is disabled and the output element is enabled. If you need to change this, refer to the description of the `kAudioOutputUnitProperty_EnableIO` property.
Recommended stream format attributes	• `kAudioFormatLinearPCM` • `AudioUnitSampleType` • `kAudioFormatFlagsAudioUnitCanonical`
Stream format notes	The outward-facing sides of the Remote I/O unit acquire their formats from the audio hardware as follows: • The input element (element 1) input scope gets its stream format from the currently-active audio input hardware. • The output element (element 0) output scope gets its stream format from the currently-active output audio hardware. Set your application format on the output scope of the input element. The input element performs format conversion between its input and output scopes as needed. Use the hardware sample rate for your application stream format. If the input scope of the output element is fed by an audio unit connection, it acquires its stream format from that connection. If, however, it is fed by a render callback function, set your application format on it.
Parameters	None in iOS.
Properties	See `I/O Audio Unit Properties`.
Property notes	You never need to set the `kAudioUnitProperty_MaximumFramesPerSlice` property on this audio unit.

Voice-Processing I/O Unit

The Voice-Processing I/O unit (subtype `kAudioUnitSubType_VoiceProcessingIO`) has the characteristics of the Remote I/O unit and adds echo suppression for two-way duplex communication. It also adds automatic gain correction, adjustment of voice-processing quality, and muting. This is the correct I/O unit to use for VoIP (Voice over Internet Protocol) apps.

All of the considerations listed in Table 3-1 (page 47) apply as well to the Voice-Processing I/O unit. In addition, there are specific properties available for this audio unit, described in `Voice-Processing I/O Audio Unit Properties`.

Generic Output Unit

Use this audio unit, of subtype `kAudioUnitSubType_GenericOutput`, when sending the output of an audio processing graph to your application rather than to the output audio hardware. You would typically use the Generic Output unit for offline audio processing. Just like the other I/O units, the Generic Output unit incorporates a Format Converter unit. This lets the Generic Output unit perform format conversion between the stream format used in an audio processing graph and the format you want.

You can also use a Generic Output unit as the final node in a subgraph that you place into a parent audio processing graph.

Using Mixer Units

iOS provides two mixer units. In most cases, you should use the Multichannel Mixer unit, which provides mixing for any number of mono or stereo streams. If you need the features of the 3D Mixer unit, you should very likely be using OpenAL instead. OpenAL is built on top of the 3D Mixer unit, providing equivalent performance with a simpler API that is well suited for game app development.

Multichannel Mixer Unit

The Multichannel Mixer unit (subtype `kAudioUnitSubType_MultiChannelMixer`) takes any number of mono or stereo streams and combines them into a single stereo output. It controls audio gain for each input and for the output, and lets you turn each input on or off separately. Starting in iOS 4.0, the Multichannel Mixer supports stereo panning for each input.

For sample code that shows how to use this audio unit, see the sample code project *Audio Mixer (MixerHost)*.

Table 3-2 provides usage details for this audio unit.

Table 3-2 Using the Multichannel Mixer unit

Audio unit feature	Details
Elements	One or more input elements, each of which can be mono or stereo. One stereo output element.

Audio unit feature	Details
Recommended stream format attributes	• `kAudioFormatLinearPCM` • `AudioUnitSampleType` • `kAudioFormatFlagsAudioUnitCanonical`
Stream format notes	On the input scope, manage stream formats as follows: • If an input bus is fed by an audio unit connection, it acquires its stream format from that connection. • If an input bus is fed by a render callback function, set your complete application stream format on the bus. Use the same stream format as used for the data provided by the callback. On the output scope, set just the application sample rate.
Parameters	See `Multichannel Mixer Unit Parameters`.
Properties	`kAudioUnitProperty_MeteringMode`.
Property notes	By default, the `kAudioUnitProperty_MaximumFramesPerSlice` property is set to a value of 1024, which is not sufficient when the screen locks and the display sleeps. If your app plays audio with the screen locked, you must increase the value of this property unless audio input is active. Do as follows: • If audio input is active, you do not need to set a value for the `kAudioUnitProperty_MaximumFramesPerSlice` property. • If audio input is not active, set this property to a value of 4096.

3D Mixer Unit

The 3D Mixer unit (subtype `kAudioUnitSubType_3DMixer`) controls stereo panning, playback tempo, and gain for each input, and controls other characteristics such as apparent distance to the listener. The output has an audio gain control. To get some idea of what this audio unit can do, consider that OpenAL in iOS is implemented using it.

In most cases, if you need the features of the 3D Mixer unit, your best option is to use OpenAL. For sample code that shows how to use OpenAL, see the sample code project *oalTouch*.

Table 3-3 provides usage details for this audio unit.

Table 3-3 Using the 3D Mixer unit

Audio unit feature	Details
Elements	One or more input elements, each of which is mono. One stereo output element.
Recommended stream format attributes	• `UInt16` • `kAudioFormatFlagsCanonical`
Stream format notes	On the input scope, manage stream formats as follows: • If an input bus is fed by an audio unit connection, it acquires its stream format from that connection. • If an input bus is fed by a render callback function, set your complete application stream format on the bus. Use the same stream format as used for the data provided by the callback. On the output scope, set just the application sample rate.
Parameters	See `3D Mixer Unit Parameters`.
Properties	See `3D Mixer Audio Unit Properties`. Note, however, that most of these properties are implemented only in the Mac OS X version of this audio unit.
Property notes	By default, the `kAudioUnitProperty_MaximumFramesPerSlice` property is set to a value of 1024, which is not sufficient when the screen locks and the display sleeps. If your app plays audio with the screen locked, you must increase the value of this property unless audio input is active. Do as follows: • If audio input is active, you do not need to set a value for the `kAudioUnitProperty_MaximumFramesPerSlice` property. • If audio input is not active, set this property to a value of 4096.

Using Effect Units

The iPod EQ unit (subtype `kAudioUnitSubType_AUiPodEQ`) is the only effect unit provided in iOS 4. This is the same equalizer used by the built-in iPod app. To view the iPod app's user interface for this audio unit, go to Settings > iPod > EQ. This audio unit offers a set of preset equalization curves such as Bass Booster, Pop, and Spoken Word.

You must supply your own user interface to the iPod EQ unit, as you must for any of the audio units. The *iPhoneMixerEQGraphTest* sample code project demonstrates how to use the iPod EQ unit and shows one way to provide a user interface for it.

Table 3-4 provides usage details for this audio unit.

Table 3-4 Using the iPod EQ unit

Audio unit feature	Details
Elements	One mono or stereo input element. One mono or stereo output element.
Recommended stream format attributes	• `kAudioFormatLinearPCM` • `AudioUnitSampleType` • `kAudioFormatFlagsAudioUnitCanonical`
Stream format notes	On the input scope, manage stream formats as follows: • If the input is fed by an audio unit connection, it acquires its stream format from that connection. • If the input is fed by a render callback function, set your complete application stream format on the bus. Use the same stream format as used for the data provided by the callback. On the output scope, set the same full stream format that you used for the input.
Parameters	None.
Properties	`kAudioUnitProperty_FactoryPresets` and `kAudioUnitProperty_PresentPreset`
Property notes	The iPod EQ unit provides a set of predefined tonal equalization curves as factory presets. Obtain the array of available EQ settings by accessing the audio unit's `kAudioUnitProperty_FactoryPresets` property. You can then apply a setting by using it as the value for the `kAudioUnitProperty_PresentPreset` property. By default, the `kAudioUnitProperty_MaximumFramesPerSlice` property is set to a value of 1024, which is not sufficient when the screen locks and the display sleeps. If your app plays audio with the screen locked, you must increase the value of this property unless audio input is active. Do as follows: • If audio input is active, you do not need to set a value for the `kAudioUnitProperty_MaximumFramesPerSlice` property. • If audio input is not active, set this property to a value of 4096.

Identifier Keys for Audio Units

This table provides the identifier keys you need to access the dynamically-linkable libraries for each iOS audio unit, along with brief descriptions of the audio units.

Table 3-5 Identifier keys for accessing the dynamically-linkable libraries for each iOS audio unit

Name and description	Identifier keys	Corresponding four-charcodes

Converter unit
Supports audio format conversions to or from linear PCM.
	`kAudioUnitType_FormatConverter`
	`kAudioUnitSubType_AUConverter`
	`kAudioUnitManufacturer_Apple`
		`aufc`
		`conv`
		`appl`

iPod Equalizer unit
Provides the features of the iPod equalizer.
	`kAudioUnitType_Effect`
	`kAudioUnitSubType_AUiPodEQ`
	`kAudioUnitManufacturer_Apple`
		`aufx`
		`ipeq`
		`appl`

3D Mixer unit
Supports mixing multiple audio streams, output panning, sample rate conversion, and more.
	`kAudioUnitType_Mixer`
	`kAudioUnitSubType_AU3DMixerEmbedded`
	`kAudioUnitManufacturer_Apple`
		`aumx`
		`3dem`
		`appl`

Multichannel Mixer unit
Supports mixing multiple audio streams to a single stream.
	`kAudioUnitType_Mixer`
	`kAudioUnitSubType_MultiChannelMixer`
	`kAudioUnitManufacturer_Apple`
		`aumx`
		`mcmx`
		`appl`

Generic Output unit
Supports converting to and from linear PCM format; can be used to start and stop a graph.
	`kAudioUnitType_Output`
	`kAudioUnitSubType_GenericOutput`
	`kAudioUnitManufacturer_Apple`
		`auou`
		`genr`
		`appl`

Remote I/O unit
Connects to device hardware for input, output, or simultaneous input and output.
	`kAudioUnitType_Output`
	`kAudioUnitSubType_RemoteIO`
	`kAudioUnitManufacturer_Apple`
		`auou`
		`rioc`
		`appl`

Voice Processing I/O unit
Has the characteristics of the I/O unit and adds echo suppression for two-way communication.
	`kAudioUnitType_Output`
	`kAudioUnitSubType_VoiceProcessingIO`
	`kAudioUnitManufacturer_Apple`
		`auou`
		`vpio`
		`appl`

Document Revision History

This table describes the changes to *Audio Unit Hosting Guide for iOS*.

Date	Notes
2010-09-01	Major revision, including addition of a chapter on using specific audio units.
2010-06-06	New document that explains how to use the system-supplied audio processing plug-ins in iOS.

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