iZotope®

iOS AUDIO PROGRAMMING

A Guide to Developing Applications for Real-time Audio Processing and Playback

www.izotope.com/iOS

IOS AUDIO PROGRAMMING GUIDE

A Guide to Developing Applications for Real-time Audio Processing and Playback

iZotope, Inc.

iZotope iOS Audio Programming Guide

This document is meant as a guide to developing applications for real-time audio processing and playback in iOS, using Core Audio services. This guide presents a broad overview of audio in iOS, as well as specific details regarding the implementation of various audio programming strategies. Important terms to be familiar with are shown in **bold** and defined in the glossary at the back of this document. Code samples and names of files containing code will be shown in

`this italicized font`

Code is taken from Apple's sample iOS applications as well as the iZotope audio effect sample application, distributed with each iZotope iOS SDK.

Who is this guide for?

This guide is meant to be a resource for developers who are new to Core Audio on iOS. It will be most useful to programmers who are experienced in iOS development but not in audio technology, or to audio programmers who are not experienced in iOS development. As such, it is highly recommended the reader have an understanding of basic concepts of both iOS development and digital signal processing before consulting this guide. For more information on DSP, or iOS development and audio programming, take a look at the **Appendix - Introductory DSP Tutorial** and **Additional Help and Resources** sections of this guide to find useful information, as well as links to other resources on the subject.

Introduction

Audio processing in iOS is done through Core Audio. Within Core Audio, there are two major strategies that allow an application to process audio data directly and play it back through the iPhone or iPad hardware, and both will be covered in detail in the main body of this document.

1. **Audio queues** are consecutive **buffers** of audio that are passed to the hardware and then reused. There are typically three buffers in a queue, and processing is done via a **callback** function, which will provide a buffer of audio that needs to be processed in time for it to be output to the iPhone/iPad hardware.

2. **Audio units** are components like mixers, equalizers, and input/output units, which take audio input and produce audio output. In iOS, built-in audio units are connected in **audio processing graphs**, wired together in the same way hardware components are connected. An audio input is provided to the processing graph, which then processes the audio before handing it off to the hardware.

The above methods do all the work of interacting with hardware and audio drivers so that an app's code can use a simple interface to handle audio, and both methods have several qualities in common.

1. Working with digital audio on any system requires working knowledge of the basic topics of **digital signal processing**. For more information on DSP, take a look at the **Appendix - Introductory DSP Tutorial** section of this document, which covers most of the basics.

2. **Audio format information** is also crucial. The **channel count**, **sampling rate**, **bit depth**, and other formatting information are all contained in audio files along with the audio data itself. When working with audio data in iOS, this format information is critical for processing and playing sound accurately. Apple provides a structure for communicating this information in the `AudioStreamBasicDescription` struct, and the strategies described in this guide rely heavily on this struct. Much of the work that goes into handling audio in iOS is in designing code that can correctly deal with a variety of audio formats.

3. Processing digital audio requires working with **buffers**. A buffer is just a short segment of **samples** whose length is convenient for processing, usually in a size that is a power of two. Buffers are time efficient because processing functions are called only when a buffer is needed, and they're space efficient because they don't occupy a huge amount of memory. The strategies for audio processing in iOS involve gathering buffers of audio from audio files or microphone input, and providing individual buffers to objects that will pass them off to the hardware.

4. An application will provide buffers whenever they are requested using a **callback**. Callbacks for audio input will have a few arguments, including a pointer to a buffer to be filled and an object or struct containing additional information that is useful for processing. This is where the digital audio is handed off, and will be passed to the audio hardware. Due to having direct access to a buffer of audio within the callback, it is possible to perform sound synthesis or processing at the individual sample level. This is also where any iZotope audio processing will take place.

This document explains audio queues and audio units by detailing the implementation of an audio playback application with both strategies. Other strategies for audio in iOS do not allow direct manipulation of audio at the sample level, so they won't be covered here. These strategies include simple frameworks for playing and recording sounds, like *`AVAudioPlayer`* and *`MPMediaPlayer`*. There is also a more complex framework for rendering 3D sound environments, called OpenAL, which is also not covered in detail within this document.

1. Audio Queues 1

i. Introduction to Audio Queue Services...1

ii. Designing a Helper Class 2

iii. Initialization 5

iv. Setup and Cleanup...8

v. Audio Queue Callback Functionality 17

vi. Running the Audio Queue 21

2. Audio Units 23

i. Introduction to Audio Units 23

ii. The MixerHostAudio Class 24

iii. Initialization 29

iv. Audio Unit Input Callback Functionality 44

v. Setting Audio Unit Parameters 47

vi. Running the Audio Processing Graph 49

vii. aurioTouch 51

3. Glossary 57

4. Additional Help and Resources 61

5. Appendix - Introductory DSP Tutorial 63

1. Audio Queues

i. Introduction to Audio Queue Services

An **audio queue** is an object used in Mac OS and iOS for recording and playing audio. It is a short queue of buffers that are constantly being filled with audio, passed to the audio hardware, and then filled again. An audio queue fills its buffers one at a time and then does the work of turning that data into sound. Audio queues handle interaction with codecs and hardware so that an app's code can interact with audio at a "high" level.

The best resource for understanding audio queue programming is the Audio Queue Services Programming Guide in the Mac OS X Developer Library. Reading that document for a full understanding of Audio Queue Services is highly recommended for anyone planning to use audio in an iOS application. Additionally, documentation of specific functions can be found in the Audio Queue Services Reference, and a list of further references can be found in the last section of this chapter.

Audio queues are defined in *AudioQueue.h* in the **AudioToolbox** framework. Most of the functions explained here are defined in *AudioQueue.h* or in *AudioFile.h*. Functions in *AudioQueue.h* have the prefix *AudioQueue*, and functions in *AudioFile.h* have the prefix *AudioFile*.

Why Use Audio Queue Services?

Audio queues have a number of benefits that make them a good choice for audio processing.

1. They are the highest level structure that allows direct processing of audio.
2. They are well documented and straightforward to use.
3. They allow precise scheduling and synchronized playback of sounds.

However, they have higher **latency** and are not as versatile as audio units.

ii. Designing a Helper Class

The easiest way to work with audio queue services in iOS is to create an Objective-C, C++ or Objective-C++ class in a project that performs all interactions with audio queues. An object needs several components to make this work properly, including interface functions, variables, initialization functions, and a callback function. This example uses a C++ class, *`AQPlayer`*, to handle all the audio queue functionality of its application. Bear in mind that this class is simply one way to get up and running with Audio Queue Services, not the only way. However, this is a good starting point for anyone who has never worked with audio queues before to begin developing powerful audio applications.

Interface

The *`AQPlayer`* class will have simple public member functions for interaction with the rest of the application.

1. The class will create and destroy *`AQPlayer`* objects with a constructor *`AQPlayer()`* and a destructor ~*`AQPlayer()`*.
2. To initialize an audio queue, there is a function *`CreateQueueForFile()`* that takes a path to an audio file in the form of a *`CFURLRef`*.
3. To start, stop, and pause audio queues, there are the functions *`StartQueue()`*, *`StopQueue()`*, and *`PauseQueue()`*.
4. To dispose of an audio queue, there is the *`DisposeQueue()`* function.
5. There will also be various functions to set and return the values of member variables, which keep track of the state of the audio queue.

Member Variables

To keep track of state in AQPlayer, several variables are required. Apple suggests using a custom struct that contains these, but in this example it is easier to just make them member variables of the C++ class *`AQPlayer`*. For clarity, all member variables will start with *m*. Variables of the following types will likely be needed for an app dealing with audio data:

```
AudioStreamBasicDescription - mDataFormat;
AudioQueueRef - mQueue;
AudioQueueBufferRef - mBuffers[kNumberBuffers];
AudioFileID - mAudioFile;
SInt64 - mCurrentPacket;
UInt32 - mNumPacketsToRead;
AudioStreamPacketDescription - *mPacketDescs;
bool - mIsDone;
```

Here's a brief description of each:

`AudioStreamBasicDescription mDataFormat` – The audio data format of the file being played. This includes information such as sample rate, number of channels, and so on.

`AudioQueueRef mQueue` – A reference to the audio queue being used for playback.

`AudioQueueBufferRef mBuffers[kNumberBuffers]` – The buffers used by the audio queue. The standard value for `kNumberBuffers` is 3, so that one buffer can be filled as another is in use, with an extra in case there's some lagging. In the callback, the audio queue will pass the individual buffer it wants filled. `mBuffers` is just used to allocate and prime the buffers. This is proof that the audio queue is literally just a queue of buffers that get filled up and wait in line to be output to the hardware.

`AudioFileID mAudioFile` – The file being played from.

`SInt64 mCurrentPacket` – The index of the next packet of audio in the audio file. This will be incremented each time playback is called.

`UInt32 mNumPacketsToRead` – The number of packets to read each time playback is called. This is also calculated after the audio queue is created.

`AudioStreamPacketDescription *mPacketDescs` – For variable bit rate (VBR) data, this is an array of packet descriptions for the audio file. For constant bit rate files, you can make this *NULL*.

`bool mIsDone` – Whether the audio queue is currently running.

It may be helpful to keep track of other information, such as whether the queue is initialized, whether it should loop playback, and so on. The only thing the constructor, `AQPlayer(),` needs to do is initialize these member variables to appropriate values.

Audio Queue Setup

Specific code is required to open audio files and to create and set up audio queues. The general process consists of:

1. Opening an audio file to determine its format.
2. Creating an audio queue.
3. Calculating buffer size.
4. Allocating buffers.
5. Some additional setup for the audio queue, like setting magic cookies, channel layout, and volume.

The details of the audio queue setup are covered in the **Initialization** section of this document, and other setup is described in the **Setup and Cleanup** section.

Audio Queue Callback

The final requirement is an audio queue callback function that the audio queue will call when it needs a buffer filled. The callback is where audio is handed off from the application to the audio queue. To keep things simple, it makes sense to implement it as a member function of the C++ `AQPlayer` class, but in general the callback can be any C function. One of the arguments to the callback is a `void*` which can be used to point to any struct or object that holds the variables for managing state. In this example, the `void*` argument will be a pointer to the `AQPlayer` object to allow access its member variables. The details of what the callback does are covered in the **Audio Queue Callback Functionality** section of this document.

iii. Initialization

Initialization functions are needed to create an audio queue for a given audio file. In this case, initialization can be performed in a member function of the `AQPlayer` class, `CreateQueueForFile()`.

Opening an Audio File

Before working with an audio queue, an audio file needs to be opened so its format can be provided when the audio queue is created. To get the format, do the following:

1. Create a `CFURLRef` that points to the file. One way to do this is with `CFURLCreateFromFileSystemRepresentation`. A typical call looks like this:

    ```
    CFURLRef audioFileURL=
    CFURLCreateFromFileSystemRepresentation( NULL,
    filePath, strlen( filePath ), false );
    ```

    Variable descriptions:
    `NULL` – The function will use the default memory allocator.

    `filePath` – A `char*` with the full path of the audio file.

    `strlen( filePath )` – The length of the path.

    `false` – The path is not for a directory, but for a single file.

    In this example, a `CFURLRef` is created outside the `AQPlayer’s` member functions, and then passed to `CreateQueueForFile()`.

2. Open the file using `AudioFileOpenURL()`. Here is a typical call:

    ```
    OSStatus result= AudioFileOpenURL( audioFileURL,
    fsRdPerm, 0, &mAudioFile );
    ```

Variable descriptions:
`audioFileURL` – The `CFURLRef` that was just created.

`fsRdPerm` – Requests read permission (rather than write, read/write, etc.) for the file.

`0` – Don't use a file type hint.

`&mAudioFile` – A pointer to an `AudioFileID` that will represent the file.

After finishing using the file's `CFURLRef`, it is important to release it using `CFRelease( audioFileURL )` to prevent a memory leak.

3. Get the file's audio data format using `AudioFileGetProperty()`. Here is a typical call:

```
UInt32 dataFormatSize= sizeof( mDataFormat );
AudioFileGetProperty( mAudioFile,
kAudioFilePropertyDataFormat, &dataFormatSize,
&mDataFormat );
```

Variable descriptions:
`mAudioFile` – The `AudioFileID` for the file, obtained from `AudioFileOpenURL()`.

`kAudioFilePropertyDataFormat` – The desired property is the data format.

`&dataFormatSize` – A pointer to an integer containing the size of the data format.

`&mDataFormat` – A pointer to an `AudioStreamBasic Description`. This has information like the sample rate, number of channels, and so on.

Creating the Audio Queue

Once the audio file's format is known, its possible to create an audio queue. For playback, `AudioQueueNewOutput()` is used (A different function is used with audio queues for recording). Here's what a call looks like:

```
AudioQueueNewOutput( &mDataFormat,
AQBufferCallback,
this, CFRunLoopGetCurrent(),
kCFRunLoopCommonModes, 0, &mQueue );
```

Variable descriptions:
`&mDataFormat` – A pointer to the data format of the file being played. The audio queue will be initialized with the right file type, number of channels, and so on.

`AQBufferCallback` – The callback function. After the function is registered here and the audio queue is started, the queue will begin to call the callback.

`this` – The `AQPlayer` object. It is passed here so that it will be passed as the `void*` argument to our callback. Note that `this` is a C++ keyword; using an Objective-C object, `self` would be used here, and using a C struct the pointer would be passed to the struct.

`CFRunLoopGetCurrent()` – Get the run loop from the current thread so that the audio queue can use it for its callbacks and any listener functions.

`kCFRunLoopCommonModes` – Allows the callback to be invoked in normal run loop modes.

`0` – Reserved by Apple. Must be 0.

`&mQueue` – A pointer to the audio queue being initialized.

Setup

After creating the audio queue, the `CreateQueueForFile()` function will call `SetupNewQueue()` to perform some additional setup, as explained in the **Setup and Cleanup** section.

iv. Setup and Cleanup

Before using the audio queue, some setup is required. Again, a member function of the `AQPlayer` class, `SetupNewQueue()` can be used, along with a few helper functions.

Setting the Buffer Size

Start by calculating the buffer size, in bytes, desired for each audio queue buffer, along with the corresponding number of packets to read from the audio file each time the callback is called. Buffers are just arrays of audio samples, and buffer size is an important value in determining the efficiency of an audio processing application. For live audio processing, a small buffer size is typically used (around 256 to 2048 bytes.) This minimizes **latency**, the time delay caused by processing each buffer. On the other hand, the shorter the buffer, the more frequently the callback will need to be called, so a balance must be struck.

To find a good buffer size, an approximate maximum packet size can be obtained from the audio file using `AudioFileGetProperty()`, and then a function to calculate the buffer size for a given duration can be created.

Write a function similar to `CalculateBytesForTime()` to calculate the buffer size. It is possible to just set a buffer size regardless of file format, but it makes sense to use a buffer size based on the packet size if the audio file uses packets. This will be another member function of the AQPlayer class. Here’s what the definition looks like:

```
void AQPlayer::CalculateBytesForTime
( AudioStreamBasicDescription &inDesc, UInt32
inMaxPacketSize, Float64 inSeconds, UInt32
*outBufferSize, UInt32 *outNumPacketsToRead )
{
…
}
```

Variable descriptions:
`&inDesc` – A pointer to the file's audio format description.

`inMaxPacketSize` – An upper bound on the packet size.

`inSeconds` – The duration in seconds for the buffer size to approximate.

*`outBufferSize` – The buffer size being returned.

*`outNumPacketsToRead` – The number of packets to read each time the callback is called, which will also calculated.

Here's what is done in the body of the function.

1. First, a reasonable maximum and minimum buffer size is chosen. For live audio processing, small values like 2048 and 256 bytes are used:

   ```
   static const int maxBufferSize = 0x800;
   static const int minBufferSize = 0x100;
   ```

2. Next, we see if the audio format has a fixed number of frames per packet and set the buffer size appropriately if it does. If it doesn't, just use a default value for buffer size from the maximum buffer size or maximum packet size:

```
if( inDesc.mFramesPerPacket != 0 ) {
Float64 numPacketsForTime=
inDesc.mSampleRate / inDesc.mFramesPerPacket *
inSeconds;
*outBufferSize= numPacketsForTime *
inMaxPacketSize;
} else {
*outBufferSize= maxBufferSize >
inMaxPacketSize ?
maxBufferSize : inMaxPacketSize;
}
```

3. Then limit the buffer size to our maximum and minimum:

```
if( *outBufferSize > maxBufferSize &&
*outBufferSize > inMaxPacketSize ) {
*outBufferSize= maxBufferSize;
} else {
     if( *outBufferSize < minBufferSize )
*outBufferSize= minBufferSize;
}
```

4. Finally calculate the number of packets to read in each callback.

```
*outNumPacketsToRead= *outBufferSize /
inMaxPacketSize;
```

Next, get the upper bound for packet size using *AudioFileGetProperty()*. The call looks like this:

```
AudioFileGetProperty( mAudioFile,
kAudioFilePropertyPacketSizeUpperBound, &size,
&maxPacketSize );
```

Variable descriptions:
`mAudioFile` – The `AudioFileID` for the file.

`kAudioFilePropertyPacketSizeUpperBound` – The desired property is the packet size upper bound, a conservative estimate for the maximum packet size.

`&size` – A pointer to an integer with the size of the desired property (set to `sizeof( maxPacketSize )`).

`&maxPacketSize` – A pointer to an integer where the maximum packet size will be stored on output.

Use `CalculateBytesForTime` to set the buffer size and number of packets to read. Here's what the call looks like:

```
UInt32 bufferByteSize;

CalculateBytesForTime( mDataFormat, maxPacketSize,
kBufferDurationSeconds, &bufferByteSize,
&mNumPacketsToRead );
```

Variable descriptions:
`mDataFormat` – The audio data format of the file being played.

`maxPacketSize` – The maximum packet size found with `AudioFileGetProperty()`.

`kBufferDurationSeconds` – A set duration for the buffer to match, approximately. In this example, the duration of the maximum buffer size is used (the buffer size in bytes divided by the sample rate).

`&bufferByteSize` – The buffer size in bytes to be set.

`mNumPacketsToRead` – The member variable where the number of packets for the callback function to read from the audio file is being stored.

Allocating Buffers

The audio queue's buffers are allocated manually. First, it is determined whether the file is in a variable bit rate format, in which case the packet descriptions in the buffers will be needed.

```
bool isFormatVBR= ( mDataFormat.mBytesPerPacket ==
0 || mDataFormat.mFramesPerPacket == 0 );

UInt32 numPacketDescs= isFormatVBR ?
mNumPacketsToRead : 0;
```

Then the audio queue's buffers are looped through and each one is allocated using `AudioQueueAllocateBufferWithPacketDescriptions()`, as follows:

```
for( int iBuf= 0; iBuf < kNumberBuffers; ++iBuf )
{
    AudioQueueAllocateBufferWithPacketDescriptions
( mQueue, bufferByteSize, numPacketDescs, mBuffers
[iBuf] );
}
```

Variable descriptions:
`mQueue` – The audio queue.

`bufferByteSize` – The buffer size in bytes calculated above.

`numPacketDescs` – 0 for CBR (constant bit rate) data, otherwise the number of packets per call to the callback.

`mBuffers[i]` - The current audio queue buffer.

`kNumberBuffers` is the number of audio queue buffers. This has already been set to 3.

If VBR (variable bit rate) data won't be used, it is possible to just use `AudioQueueAllocateBuffer()`, which takes all the same arguments except for `numPacketDescs`.

Audio File Properties

Some audio files have additional properties that provide necessary information to an audio queue, such as magic cookies and channel layout.

Setting a magic cookie

Some compressed audio formats store information about the file in a structure called a magic cookie. If the file being played has a magic cookie, it needs to be provided to the audio queue. The following code will check if there is a magic cookie and give it to the audio queue if there is.

First, check whether the audio file has a magic cookie property:

```
size= sizeof( UInt32 ); OSStatus result=
AudioFileGetPropertyInfo( mAudioFile,
kAudioFilePropertyMagicCookieData, &size, NULL );
```

If it does, get the property from the audio file and set it in the audio queue:

```
if( result == noErr && size > 0 ) {
     char* cookie= new char [size];
     AudioFileGetProperty( mAudioFile,
     kAudioFilePropertyMagicCookieData, &size,
     cookie );
     AudioQueueSetProperty( mQueue,
     kAudioQueueProperty_MagicCookie, cookie,
     size );
     delete [] cookie;
}
```

Variable descriptions:
mAudioFile – The audio file.

kAudioFilePropertyMagicCookieData – The desired property is magic cookie data.

&size – Pointer to an integer with the size of the desired property.

cookie – Temporary storage for the magic cookie data.

mQueue – The audio queue.

kAudioQueueProperty_MagicCookie – The desired property to set is the queue's magic cookie.

Setting a channel layout

If an audio file has a channel layout property, the channel layout of the audio queue needs to be set. The channel layout property is normally used only for audio with more than two channels.

First, check whether the audio file has a channel layout property:

```
size= sizeof( UInt32 );
result= AudioFileGetPropertyInfo( mAudioFile,
kAudioFilePropertyChannelLayout, &size, NULL );
```

If it does, get the property from the audio file and set it in the audio queue:

```
if( result == noErr && size > 0 ){
AudioChannelLayout* acl=
static_cast<AudioChannelLayout*>(malloc( size ));
AudioFileGetProperty( mAudioFile,
kAudioFilePropertyChannelLayout, &size, acl );
AudioQueueSetProperty( mQueue,
kAudioQueueProperty_ChannelLayout, acl, size );
free( acl );
}
```

Variable descriptions:
mAudioFile – The audio file.

kAudioFilePropertyChannelLayout – The desired property is the channel layout.

&size – Pointer to an integer with the size of the desired property.

acl – Temporary storage for the channel layout data.

mQueue – The audio queue.

kAudioQueueProperty_ChannelLayout – The desired property to set is the queue’s channel layout.

Property Listeners

A property listener is a function that the audio queue will call when one of its properties changes. Property listeners can be useful for tracking the state of the audio queue. For example, in this example there is a member function `isRunningProc()` that is called when the audio queue's `IsRunning` property changes. The function definition looks like this:

```
void AQPlayer::isRunningProc( void* inAQObject,
AudioQueueRef inAQ, AudioQueuePropertyID inID ) {
    …
}
```

Variable descriptions:
`void* inAQObject` – A pointer to the AQPlayer object.

`AudioQueueRef inAQ` – The audio queue calling the property listener.

`AudioQueuePropertyID inID` – A value specifying the property that has changed.

In the body of the function, the value of our AQPlayer's `mIsRunning` property is updated. A property listener can take any action that would be appropriate when a specific property changes. As done in the callback, cast `inAQObject` as a pointer to an `AQPlayer`:

```
AQPlayer* THIS= static_cast<AQPlayer*>(inAQObject);
```

Then update the *mIsRunning* property using `AudioQueueGetProperty ()`.

```
UInt32 size= sizeof( THIS->mIsRunning );

OSStatus result= AudioQueueGetProperty( inAQ,
kAudioQueueProperty_IsRunning, &THIS->mIsRunning,
&size );
```

To add the property listener to the audio queue so that it will be called, use `AudioQueueAddPropertyListener()`. That is done in the `SetupNewQueue()` function. Here's what the call looks like:

```
AudioQueueAddPropertyListener( mQueue,
kAudioQueueProperty_IsRunning, isRunningProc,
this );
```

Variable descriptions:
`mQueue` – The audio queue.

`kAudioQueueProperty_IsRunning` – The desired property to listen for is *IsRunning*.

`isRunningProc` – The property listener function created.

`this` – The `AQPlayer` object, which the audio queue will pass as the `inAQObject` argument to `isRunningProc`.

To remove a property listener, use `AudioQueueRemovePropertyListener()`, which takes the same arguments.

Setting Volume

Volume is an adjustable parameter of an audio queue, (in fact, it is the only adjustable parameter), and to set it, use the `AudioQueue SetParameter()` function as follows:

```
AudioQueueSetParameter( mQueue,
kAudioQueueParam_Volume, 1.0 );
```

Variable descriptions:
`mQueue` – The audio queue.

`kAudioQueueParam_Volume` – The parameter to set is volume. The third argument is volume as a floating point value from 0.0 to 1.0.

Cleanup

When finished with the audio queue, dispose of the queue and close the audio file. Disposing of an audio queue also disposes of its buffers. This is done in the member function `DisposeQueue()`.

```
AudioQueueDispose( mQueue, true );
```

Variable descriptions:
`mQueue` – The audio queue.

`true` – Dispose immediately, rather than playing all queued buffers first. (This is a 'synchronous' action.)

The `DisposeQueue()` function takes a boolean `inDisposeFile` specifying whether to dispose the audio file along with the queue. To dispose the audio file use: `AudioFileClose( mAudioFile );` The destructor, `~AQPlayer()`, just needs to call `DisposeQueue( true )` to dispose of the audio queue and audio file.

v. Audio Queue Callback Functionality

To provide audio data to an audio queue, a playback audio queue callback function is created. The audio queue will call this function every time it needs a buffer of audio, and it's up to the callback to provide it. Normally the callback will read directly from a file. If an application performs its own processing or synthesis, the callback will also use previously created functions to provide audio. In this example, the callback is a member function of the *AQPlayer* class. A callback declaration looks like this:

```
void AQPlayer::AQBufferCallback( void* inAQObject,
AudioQueueRef inAQ, AudioQueueBufferRef inBuffer )
{
      …
}
```

Variable descriptions:
void inAQObject* – A pointer to any struct or object used to manage state. Typically this will be either a custom C struct or an Objective-C or C++ object handling the audio queue. In these examples, this will be a pointer to the C++ *AQPlayer* object, which handles audio queues and stores all the necessary information in member variables.

AudioQueueRef inAQ – The audio queue calling the callback. The callback will place a buffer of audio on this queue.

AudioQueueBufferRef inBuffer – The buffer the audio queue is requesting be filled. Usually the callback will read from a file into the buffer and then enqueue the buffer.

Getting the Object

The first thing the app's callback should do is get a pointer to the struct or object that has all the state variables. In this case, just cast the *void* inAQObject* as a pointer to an *AQPlayer* using a static cast:

```
AQPlayer* THIS= static_cast<AQPlayer*>
(inAQObject);
```

Now, to access the AQPlayer's member variables, just use the arrow operator: *THIS->mAudioFile*.

Reading From a File

Now the audio queue buffer should be filled with audio data from a file. To read from an audio file into an audio queue buffer, use the *AudioFileReadPackets()* function. Here's what a call looks like:

```
AudioFileReadPackets( THIS->mAudioFile, false,
&numBytes, inBuffer->mPacketDescriptions, THIS-
>mCurrentPacket, &nPackets, inBuffer-
>mAudioData );
```

Variable descriptions:
`THIS->mAudioFile` – The audio file being played from.

`false` – Don't cache data.

`&numBytes` – Pointer to an integer that will give the number of bytes read.

`inBuffer->mPacketDescriptions` – The packet descriptions in the current buffer.

`THIS->mCurrentPacket` – The index of the next packet of audio.

`&nPackets` – Pointer to the number of packets to read. On output, will tell how many were actually read.

`inBuffer->mAudioData` – The audio buffer that will be filled.

Processing

After filling an audio buffer from a file, it is possible to process the audio directly by manipulating the values in the buffer. And if the application uses synthesis, not playback, it is possible to fill the buffer directly without using a file. This is also where any iZotope effects would be applied. Details regarding the implementation of iZotope effects are distributed along with our effect libraries.

Enqueueing a Buffer

After filling a buffer with audio data, check whether any packets were actually read by looking at the value of nPackets. If no packets were read, the file has ended, so stop the audio queue and set the mIsDone variable to true. Otherwise, place the filled buffer in the audio queue using the `AudioQueueEnqueueBuffer()` function. A call to this function has the following form:

```
AudioQueueEnqueueBuffer( inAQ, inBuffer,
numPackets, THIS->mPacketDescs );
```

Variable descriptions:
`inAQ` – The audio queue to place the buffer on.

`inBuffer` – The buffer to enqueue.

`numPackets` – The number of packets in the buffer. For constant bit rate audio formats, which do not use packets, this value should be 0.

`THIS->mPacketDescs` – The packet descriptions for the packets in the buffer. For constant bit rate audio formats, which do not use packets, this value should be NULL.

Ending the Callback

The callback should also stop the audio queue when the audio file ends. To do this, check the value of `nPackets` after calling `AudioFileReadPackets()`. If `nPackets` is 0, set the `mIsDone` variable to true and stop the audio queue using the `AudioQueueStop()` function as follows:

```
THIS->mIsDone= true;
AudioQueueStop( inAQ, false );
```

Variable descriptions:
`inAQ` – The audio queue being used.

`false` – Do not stop immediately; instead, play all queued buffers and then stop. (This is an ‘asynchronous’ action). (This example also checks the mIsLooping member variable and, if it is true, resets the current packet index so that the file will loop.)

vi. Running the Audio Queue

Starting the Queue

Now that all the necessary information is stored in variables within the `AQPlayer` class, along with proper initialization and a callback function, it is possible to actually start the audio queue. This is done in the member function `StartQueue()`. `StartQueue()` also checks whether there is a need to create an audio queue for the current audio file. First, set the current packet index to 0 so it starts at the beginning of the file:

```
mCurrentPacket= 0;
```

Next, prime the queue's buffers so that there is audio data in them when the queue starts. Otherwise data from uninitialized buffers will briefly be heard instead of audio.

```
for( int iBuf= 0; iBuf < kNumberBuffers; ++iBuf )
{
     AQBufferCallback( this, mQueue, mBuffers
[iBuf] );
}
```

Then start the queue using `AudioQueueStart()`:

```
AudioQueueStart( mQueue, NULL );
```

Variable descriptions:
`mQueue` – The audio queue.

`NULL` – Start immediately. It is also possible to specify a start time with an `AudioTimeStamp` here.

Pausing the Queue

It is possible to pause the audio queue by using `AudioQueuePause()`.

```
AudioQueuePause( mQueue );
```

This function will pause the audio queue provided to it. This is the only thing the `PauseQueue()` function in the `AQPlayer` does. The queue will start again when `AudioQueueStart()` is called.

Stopping the Queue

It is possible to stop the audio queue at any time with `AudioQueueStop ()`, which is done in the `AudioQueueStop()` function. The sample application simply stops the audio queue at the end of the audio file.

```
AudioQueueStop( mQueue, false );
```

Variable descriptions:
`mQueue` – The audio queue.

`false` – Do not stop immediately – instead, play all queued buffers and then stop (This is an ‘asynchronous’ action).

2. Audio Units

i. Introduction to Audio Units

Audio units are a service used for audio processing at a lower level in iOS. An individual audio unit takes audio input and produces audio output, and they may be connected into an **audio processing graph**, in which each unit passes its output to the next. For example, it is possible to pass audio to an EQ unit, the output of which is passed to a mixer unit, whose output is passed to an I/O (input/output) unit, which gives it to the hardware to turn into sound. The graph structure is similar to the layout of an audio hardware setup, which makes it easy to visualize the flow of audio in an application. Alternatively, it is possible to use just a single audio unit, or connect them without using a processing graph.

In Mac OS, audio unit programming is a much broader topic because it is possible to create audio units. iOS programming however, simply makes use of audio units that already exist. Since the audio units provide access to the low-level audio data, it is possible to process that audio in any desired way. In iOS 4, there are seven audio units available, which handle EQ, mixing, I/O, voice processing, and format conversion. Normally an app will have a very simple audio processing graph with just a few audio units.

A great resource for understanding audio unit programming for iOS is the **Audio Unit Hosting Guide** in the iOS Developer Library. This document covers the details of using audio units for iOS. (The Audio Unit Programming Guide in the Mac OS X Developer Library does not have as much useful information for iOS programming). Documentation of specific functions can be found in the Audio Unit Framework Reference.

Apple's **MixerHost** sample application is a great example of using audio units. This guide will focus on that sample code along with some discussion of other sample code like **aurioTouch**. For an example like MixerHost but with a more complex audio processing graph, take a look at the iPhoneMixerEQGraphTest example.

Audio units and other structures necessary for using them are defined in *AudioComponent.h*, *AUGraph.h* and other headers included from *AudioToolbox.h* in the AudioToolbox framework. Similarly, necessary functions for handling audio sessions are found in `AVFoundation.h` in the AVFoundation framework (an application can just include `AudioToolbox/AudioToolbox.h` and `AVFoundation/AVFoundation.h`). It should be noted that many of the functions and structures used by audio units have the AU prefix.

Why use Audio Units?

Audio units have several benefits.

1. They are the lowest latency audio processing strategy in iOS.
2. They have built-in support for EQ, mixing, and other processing.
3. They allow versatile, real-time manipulation of a flexible processing graph.

The drawbacks to audio units are the subtleties and complexities of their setup and use. In general they are not as well-documented as audio queues, and have a steeper learning curve.

ii. The MixerHostAudio Class

Just as with audio queues, the easiest way to work with audio units in iOS is to create an Objective-C, C++ or Objective-C++ class in a project that performs all interactions with audio units. Again, it is necessary to interface functions, variables, initialization functions, and a callback function, similar to the callback used by an audio queue. This example will examine the Objective-C class *MixerHostAudio* from the MixerHost sample application, which handles all the audio unit functionality of the application. Again, this class is simply one way to use audio units—it is also possible to use another C++ class to handle the audio unit interaction.

The `MixerHostAudio` class uses just two audio units in a very simple audio processing graph - mono or stereo audio is passed through a **Multichannel Mixer** audio unit to a **Remote I/O** audio unit, which provides its output to the device's audio hardware.

Interface

The *MixerHostAudio* class is a subclass of *NSObject*. So, as expected, it implements the *init* and *dealloc* methods, which handle proper initialization and destruction. It also has several instance methods, some for setup and some for interfacing with the rest of the application. The methods that are important when using a *MixerHostAudio* object, as can be seen from how it's used by the *MixerHostViewController*, are for initialization (*init*), starting and stopping the audio processing graph (*startAUGraph* and *stopAUGraph*), and setting the state and parameters of the mixer unit (*enableMixerInput:isOn:*, *setMixerOutputGain:*, and *setMixerInput:gain:*).

The other methods are used by the *MixerHostAudio* object itself for initialization are

- *obtainSoundFileURLs*
- *setupAudioSession*
- *setupStereoStreamFormat*
- *setupMonoStreamFormat*
- *readAudioFilesIntoMemory*
- *configureAndInitializeAudioProcessingGraph*

and for logging messages conveniently:

- *printASBD:*
- *printErrorMessage:*

So the interface for the *MixerHostAudio* class is simpler than it seems at first glance – initialize it, start and stop its processing graph, and set the parameters for its mixer audio unit.

Member Variables

To keep track of state in *MixerHostAudio*, there are several member variables. First, there is a custom struct (*soundStruct*) defined, which contains necessary information to pass to the callback function:

```
typedef struct {

    BOOL                isStereo;
    UInt32              frameCount;
    UInt32              sampleNumber;
    AudioUnitSampleType *audioDataLeft;
    AudioUnitSampleType *audioDataRight;

} soundStruct, *soundStructPtr;
```

Variable descriptions:
isStereo – Whether the input data has a right channel.

frameCount – Total frames in the input data.

sampleNumber – The index of the current sample in the input data.

audioDataLeft – The audio data from the left channel (or the only channel, for mono data).

audioDataRight – The audio data from the right channel (or *NULL* for mono data).

The member variables of *MixerHostAudio* are:

Float64 - graphSampleRate;

CFURLRef - sourceURLArray[NUM_FILES];

soundStruct - soundStructArray[NUM_FILES];

AudioStreamBasicDescription - stereoStreamFormat;

AudioStreamBasicDescription - monoStreamFormat;

AUGraph - processingGraph;

BOOL - playing;

BOOL - interruptedDuringPlayback;

AudioUnit - mixerUnit;

Here's a brief description of each:

Float64 graphSampleRate – The sample rate in Hz used by the audio processing graph.

CFURLRef sourceURLArray[*NUM_FILES*] – An array of URLs for
the input audio files used for playback.

soundStruct soundStructArray[*NUM_FILES*] – An array of *soundStructs*, which are defined above. One
soundStruct is kept for each file in order to keep track of all the necessary data separately.

AudioStreamBasicDescription stereoStreamFormat – The audio data format for a stereo file. Along with the number of channels, this includes information like sample rate, bit depth, and so on.

AudioStreamBasicDescription monoStreamFormat – The audio
data format for a mono file.

AUGraph processingGraph – The audio processing graph that contains the Multichannel Mixer audio unit and the Remote I/O unit. The graph does all the audio processing.

BOOL playing – Whether audio is currently playing.

BOOL interruptedDuringPlayback – Whether the audio session has been interrupted (by a phone call, for example). This flag lets allows for restarting the audio session after returning from an interruption.

`AudioUnit mixerUnit` – The Multichannel Mixer audio unit. Its member variables are kept track of in order to change its parameters later, unlike the Remote I/O unit, which doesn't need to be referenced after it is added to the processing graph.

Audio Unit Setup

Some substantial code is required to create audio units and set up the audio processing graph. These are the steps that MixerHostAudio uses for initialization:

1. Set up an active audio session.
2. Obtain files, set up their formats, and read them into memory.
3. Create an audio processing graph.
4. Create and initialize Multichannel Mixer and Remote I/O audio units.
5. Add the audio units to the graph and connect them.
6. Start processing audio with the graph.

The details of the audio unit setup are covered in the **Audio Unit Initialization** section of this document.

Audio Unit Callback

The last thing needed is a callback function that the Multichannel Mixer audio unit will call when it needs a buffer filled. The callback is where audio is handed off from the application to the mixer unit. After that point, the audio doesn't need to be dealt with again—it will pass through the Multichannel Mixer unit, through the Remote I/O unit, and to the audio hardware. The callback is implemented as a static C function in `MixerHostAudio.m`, and is *not* a method of the `MixerHostAudio` class. It must be a C function, not a method. Apple warns against making Objective-C calls from a callback, in order to keep calculations fast and efficient during audio processing. So the callback doesn't actually know anything about the MixerHostAudio object—it just takes in the data sent to it and adds samples to the buffer that the mixer unit asks it to fill.

Just as in an audio queue callback, one of the arguments to the callback is a `void`* which can be used to point to any struct or object that holds the variables for managing state. In this case, the `void`* argument is a pointer to the member variable *soundStructArray* so the necessary information that's been stored there can be accessed. The details of what the callback does are covered in the **Audio Unit Callback Functionality** section of this document.

iii. Initialization

To initialize the *MixerHostAudio* object and its audio processing graph, it is necessary to do several things. Initialization is grouped into several methods in *MixerHostAudio* to keep things organized. (For brevity, the code shown here leaves out the error checking that the sample code implements.)

Starting an Audio Session

Start an audio session using the method *setupAudioSession*. The session is an *AVAudioSession* object shared throughout the application. First get the session object, set the *MixerHostAudio* object as its delegate to receive interruption messages, and set the category of the session to Playback.

```
AVAudioSession *mySession = [AVAudioSession
sharedInstance];
[mySession setDelegate: self];
NSError *audioSessionError = nil;
[mySession setCategory:
AVAudioSessionCategoryPlayback error:
&audioSessionError];
```

Next, request the desired sample rate in the audio session. Note that it's impossible to set this value directly—all that can be done is to request it and check what the actual value is. There is never a guarantee that the preferred sample rate will be used.

```
self.graphSampleRate= 44100.0;
[mySession setPreferredHardwareSampleRate:
graphSampleRate error: &audioSessionError];
```

Now activate the audio session and then match the graph sample rate to the one the hardware is actually using.

```
[mySession setActive: YES error:
&audioSessionError]; self.graphSampleRate =
[mySession currentHardwareSampleRate];
```

Finally, register a property listener, just as was done with an audio queue. Here it is possible to detect audio route changes (like plugging in headphones) so that playback can be stopped when that happens.

```
AudioSessionAddPropertyListener
( kAudioSessionProperty_AudioRouteChange,
audioRouteChangeListenerCallback, self );
```

Variable descriptions:
kAudioSessionProperty_AudioRouteChange – The property of interest, in this case the audio route change property.

audioRouteChangeListenerCallback – A function that has been defined and will be called when the property changes.

self – A pointer to any data for the callback to take as an argument, in this case the *MixerHostAudio* object.

Property listeners

Just as with audio queues, an audio session's property listeners will be called when some property of the session changes. The function that was just registered, *audioRouteChangeListenerCallback()*, is called when the audio queue's *AudioRouteChange* property changes—e.g. when headphones are plugged into the device. The function definition looks like this:

```
void audioRouteChangeListenerCallback (
     void *inUserData, AudioSessionPropertyID
     inPropertyID, UInt32 inPropertyValueSize,
     const void    *inPropertyValue )
{
     …
}
```

Variable descriptions:
`void* inUserData` – A pointer to any struct or object containing the desired information, in this case the MixerHostAudio object.

`AudioSessionPropertyID inID` – A value specifying the property that has changed.

`UInt32 inPropertyValueSize` – The size of the property value.

`const void *inPropertyValue` – A pointer to the property value.

In the body of the function, check whether playback should stop based on what type of audio route change occurred. First, make sure this is a route change message, then get the *`MixerHostAudio`* object with a C-style cast of *inUserData*.

```
if (inPropertyID !=
kAudioSessionProperty_AudioRouteChange)
return;
MixerHostAudio *audioObject =
(MixerHostAudio *)inUserData;
```

If sound is not playing, do nothing. Otherwise, check the reason for the route change and stop playback if necessary by posting a notification that the view controller will get. There's some fairly complicated stuff going on with types here, but it won't be explored in detail.

```
if (NO == audioObject.isPlaying) {
     return;
} else {
          CFDictionaryRef routeChangeDictionary =
inPropertyValue;
CFNumberRef routeChangeReasonRef =
CFDictionaryGetValue( routeChangeDictionary, CFSTR
(kAudioSession_AudioRouteChangeKey_Reason));
SInt32 routeChangeReason;
CFNumberGetValue (routeChangeReasonRef,
kCFNumberSInt32Type, &routeChangeReason);
     if (routeChangeReason ==
kAudioSessionRouteChangeReason_OldDeviceUnavailabl
e) {

NSString*MixerHostAudioObjectPlaybackStateDidChang
eNotification
=@"MixerHostAudioObjectPlaybackStateDidChangeNotif
ication"; [[NSNotificationCenter defaultCenter]
postNotificationName:
MixerHostAudioObjectPlaybackStateDidChangeNotifica
tion object: self];
     }
     }
```

Obtaining Sound File URLs

Next, use the method `obtainSoundFileURLs` to get the URLs to the sound files wanted for playback. All that's needed is to get an `NSURL`* using the path and then store it as a `CFURLRef` in an array for later use (*retain* is used so the reference won't be autoreleased).

```
NSURL *guitarLoop= [[NSBundle mainBundle]
URLForResource:@"guitarStereo" withExtension:
@"caf"];
NSURL *beatsLoop= [[NSBundle mainBundle]
URLForResource:@"beatsMono" withExtension:@"caf"];
sourceURLArray[0] = (CFURLRef) [guitarLoop
retain]; sourceURLArray[1] = (CFURLRef) [beatsLoop
retain];
```

Setting Up Mono and Stereo Stream Formats

Now set up the `AudioStreamBasicDescription` member variables for each desired playback format, in the methods `setupStereoStreamFormat` and `setupMonoStreamFormat`. Notice that these methods are identical except for what they set as `mChannelsPerFrame` and what they log. It would be easy to collapse these into a single method, but presumably they're here for the sake of clarity. It is also possible to take the audio data formats from the files themselves if accepting different formats was desired, as was done in the audio queue example, but this sample application is only designed to work with the files it has built in.

Take a look at `setupStereoStreamFormat`. First, get the size in bytes of `AudioUnitSampleType`. This type is defined to be a signed 32-bit integer in `CoreAudioTypes.h`, so it has 4 bytes.

```
size_t bytesPetSample = sizeof (AudioUnitSampleType);
```

Then set all the values in the `AudioStreamBasicDescription`.

```
stereoStreamFormat.mFormatID =
kAudioFormatLinearPCM;
stereoStreamFormat.mFormatFlags =
kAudioFormatFlagsAudioUnitCanonical;
stereoStreamFormat.mBytesPerPacket   =
bytesPerSample;
stereoStreamFormat.mFramesPerPacket = 1;
stereoStreamFormat.mBytesPerFrame =
bytesPerSample;
stereoStreamFormat.mChannelsPerFrame = 2;
stereoStreamFormat.mBitsPerChannel   = 8 *
bytesPerSample;
stereoStreamFormat.mSampleRate = graphSampleRate;
```

Variable descriptions:
`mFormatID` – An identifier tag for the format. Linear PCM (pulse-code modulation) is very common—it's the format for .wav and .aif files, for example. Here it is used for the .caf files Apple has included.

mFormatFlags – Here it is possible to specify some flags to hold additional details about the format. In this case, the canonical audio unit sample format is being used to avoid extraneous conversion.

mBytesPerPacket – The number of bytes in each packet of audio. A packet is just a small chunk of data whose size is determined by the format and is the smallest 'readable' piece of data. For uncompressed linear PCM data (which is the data type here) each packet has just one sample, so this is set to the number of bytes per sample.

mFramesPerPacket – The number of frames in each packet of audio. A frame is just a group of samples that happen at the same instant—so in stereo, there are two samples in each frame. Again, for uncompressed audio this value is one. Just read frames one at a time from the file.

mBytesPerFrame – The number of bytes in each frame. Even though each frame has two samples in stereo, this will again be the number of bytes per sample. This is because the data is **noninterleaved** – each channel is held in a separate array, rather than having a single array in which samples alternate between the two channels.

mChannelsPerFrame – The number of channels in the audio – one for mono, two for stereo, etc.

mBitsPerChannel – The bit depth of each channel—how many bits represent a single sample. Using *AudioUnitSampleType*, just convert from bytes to bits by multiplying by 8.

mSampleRate – The sample rate of the format in frames per second (Hz.) This is the value obtained from the hardware, which may or may not be the preferred sample rate.

Reading Audio Files Into Memory

Next, read the audio files into memory so that audio data can be used from within the callback, with the `readAudioFilesIntoMemory` method. Note that this is a very different strategy than in the audio queue example, in which the callback opened and read directly from a file each time it needed to fill a buffer. The two methods have benefits and drawbacks. Reading the files into memory means there isn't a need to keep opening the file, which will make the callback more efficient. But it means space needs to be allocated for the audio data, which will use up a lot of memory if the files are long.

Loop through the same process for each audio file, so all the code in this method is inside this *for* loop:

```
for(int audioFile = 0; audioFile < NUM_FILES;

     ++audioFile) {

…

}
```

Next, open a file. The audio queue example used `AudioFile.h`. Here `ExtendedAudioFile.h` is used, which just handles additional encoding and decoding. First create an `ExtAudioFileRef` and load the file into it from the URL obtained earlier.

```
ExtAudioFileRef audioFileObject = 0;
OSStatus result = ExtAudioFileOpenURL
( sourceURLArray[audioFile], &audioFileObject);
```

Getting properties

1. First get the number of frames in the file so that memory can be properly allocated for the data, and so it's possible to keep track of location when reading it. Store the value in the `soundStruct` that corresponds with the file.

```
UInt64 totalFramesInFile = 0;
UInt32 frameLengthPropertySize = sizeof
(totalFramesInFile);
result = ExtAudioFileGetProperty
( audioFileObject,
kExtAudioFileProperty_FileLengthFrames,
&frameLengthPropertySize, &totalFramesInFile);
soundStructArray[audioFile].frameCount =
totalFramesInFile;
```

Variable descriptions:
`audioFileObject` – The file, `ExtAudioFileRef` from above.

`kExtAudioFileProperty_FileLengthFrames` – The property of interest, in this case file length in frames.

`&frameLengthPropertySize` – Pointer to the size of the property of interest.

`&totalFramesInFile` – Pointer to the variable where this value should be stored.

2. Now get the channel count in the file for correct playback. Store the value in a variable *channelCount*, which will be used shortly.

```
AudioStreamBasicDescription fileAudioFormat = {0};
UInt32 formatPropertySize = sizeof
(fileAudioFormat);
result = ExtAudioFileGetProperty (audioFileObject,
kExtAudioFileProperty_FileDataFormat,
&formatPropertySize, &fileAudioFormat);
UInt32 channelCount=
fileAudioFormat.mChannelsPerFrame;
```

Variable descriptions:
`audioFileObject` – The file, `ExtAudioFileRef` from above.

`kExtAudioFileProperty_FileDataFormat` – The property of interest, in this case the data format description.

`&formatPropertySize` – Pointer to the size of the property of interest.

`&fileAudioFormat` – Pointer to the variable where this value should be stored, here the `AudioStreamBasicDescription`.

Allocating memory

Now that the channel count is known, memory can be allocated to store the audio data.

1. First, allocate the left channel, whose size is just the number of samples times the size of each sample.

   ```
   soundStructArray[audioFile].audioDataLeft =
   (AudioUnitSampleType *) calloc (totalFramesInFile,
   sizeof(AudioUnitSampleType));
   ```

2. Next, if the file is stereo, allocate the right channel in the same way. Also set the `isStereo` value in the `soundStruct` and store the appropriate audio data format in a variable `importFormat`, which will be used below.

   ```
   AudioStreamBasicDescription importFormat = {0};
   if (2 == channelCount) {
        soundStructArray[audioFile].isStereo = YES;
        soundStructArray[audioFile].audioDataRight =
        (AudioUnitSampleType *) calloc
        (totalFramesInFile, sizeof
        (AudioUnitSampleType));
        importFormat = stereoStreamFormat;
   } else if (1 == channelCount) {
        soundStructArray[audioFile].isStereo = NO;
        importFormat = monoStreamFormat;
   }
   ```

3. Use the *importFormat* variable to set the data format of the *audioFileObject* so that it will use the correct format when the audio data is copied into each file's *soundStruct*.

```
result = ExtAudioFileSetProperty
( audioFileObject,
kExtAudioFileProperty_ClientDataFormat, sizeof
(importFormat), &importFormat );
```

Setting up a buffer list

To read the audio files into memory, first set up an *AudioBufferList* object. As Apple's comments explain, this object gives the *ExtAudioFileRead* function the correct configuration for adding data to the buffer, and it points to the memory that should be written to, which was allocated in the *soundStruct*.

1. First, allocate the buffer list and set its channel count.

```
AudioBufferList *bufferList;
bufferList = (AudioBufferList *) malloc ( sizeof
(AudioBufferList) + sizeof (AudioBuffer) *
(channelCount - 1);
bufferList->mNumberBuffers = channelCount;
```

2. Create an empty buffer as a placeholder and place it at each index in the buffer array.

```
AudioBuffer emptyBuffer = {0};
size_t arrayIndex;
for (arrayIndex = 0; arrayIndex < channelCount;
arrayIndex++) {
     bufferList->mBuffers[arrayIndex] =
emptyBuffer;
     }
```

3. Finally, set the properties of each buffer—the channel count, the size of the data, and where the data will actually be stored.

```
bufferList->mBuffers[0].mNumberChannels = 1;
bufferList->mBuffers[0].mDataByteSize =
totalFramesInFile * sizeof (AudioUnitSampleType);
bufferList->mBuffers[0].mData = soundStructArray
[audioFile].audioDataLeft;
     if (2 == channelCount) {
          bufferList->mBuffers[1].mNumberChannels =
1;
          bufferList->mBuffers[1].mDataByteSize =
          totalFramesInFile * sizeof
          (AudioUnitSampleType);
          bufferList->mBuffers[1].mData =
          soundStructArray
          [audioFile].audioDataRight;
}
```

Reading the data

With the buffers all set up and ready to be filled, the audio data can be read in from the audio files using `ExtAudioFileRead`.

```
UInt32 numberOfPacketsToRead = (UInt32
totalFramesInFile;
result = ExtAudioFileRead ( audioFileObject,
&numberOfPacketsToRead, bufferList );
free (bufferList);
```

After that, along with error checking, all that needs to be done is set the sample index to 0 and get rid of `audioFileObject`.

```
soundStructArray[audioFile].sampleNumber = 0;
ExtAudioFileDispose (audioFileObject);
```

Setting Up an Audio Processing Graph

The rest of the initialization will take place in the `configureAndInitializeAudioProcessingGraph` method. Here an audio processing graph will be created and audio units will be added to it. (Again, error checking code is not reproduced here.)

1. First create an `AUGraph`:

   ```
   result = NewAUGraph (&processingGraph);
   ```

2. Next, set up individual audio units for use in the graph. Set up a component description for each one, the Remote I/O unit and the Multichannel Mixer unit.

   ```
   AudioComponentDescription iOUnitDescription;

   iOUnitDescription.componentType =
   kAudioUnitType_Output;

   iOUnitDescription.componentSubType =
   kAudioUnitType_RemoteIO;

   iOUnitDescription.componentManufacturer =

   kAudioUnitManufacturer_Apple;

   iOUnitDescription.componentFlags = 0;

   iOUnitDescription.componentFlagsMask = 0;

   AudioComponentDescription MixerUnitDescription;

   MixerUnitDescription.componentType =
   kAudioUnitType_Mixer;

   MixerUnitDescription.componentSubType =
   kAudioUnitType_MultiChannelMixer;

   MixerUnitDescription.componentManufacturer =

   kAudioUnitManufacturer_Apple;

   MixerUnitDescription.componentFlags = 0;
   ```

```
MixerUnitDescription.componentFlagsMask = 0;
```

Variable descriptions:
`componentType` – The general type of the audio unit. The project has an output unit and a mixer unit.

`componentSubType` – The specific audio unit subtype. The project has a Remote I/O unit and a Multichannel Mixer unit.

`componentManufacturer` – Manufacturer of the audio unit. All the audio units usable in iOS are made by Apple.

`componentFlags` – Always 0.

`componentFlagsMask` – Always 0.

3. Then add the nodes to the processing graph. `AUGraphAddNode` takes the processing graph, a pointer to the description of a node, and a pointer to an `AUNode` in which to store the added node.

```
AUNode iONode;
AUNode mixerNode;

result = AUGraphAddNode ( processingGraph,
&iOUnitDescription, &iONode );
result = AUGraphAddNode ( processingGraph,
&MixerUnitDescription, &mixerNode );
```

4. Open the graph to prepare it for processing. Once it's been opened, the audio units have been instantiated, so it's possible to retrieve the Multichannel Mixer unit using `AUGraphNodeInfo`.

```
result = AUGraphOpen (processingGraph);
result = AUGraphNodeInfo ( processingGraph,
mixerNode, NULL, &mixerUnit );
```

Setting up the mixer unit

Some setup needs to be done with the mixer unit.

1. Set the number of buses and set properties separately for the guitar and beats buses.

   ```
   UInt32 busCount = 2;
   UInt32 guitarBus = 0;
   UInt32 beatsBus = 1;
   result = AudioUnitSetProperty ( mixerUnit,
   kAudioUnitPropertyElementCount,
   kAudioUnitScopeInput, 0, &busCount, sizeof
   (busCount));
   ```

   Variable descriptions:
   mixerUnit – The audio unit a property is being set in.

   kAudioUnitPropertyElementCount – The desired property to set is the number of elements.

   kAudioUnitScopeInput – The scope of the property. Audio units normally have input scope, output scope, and global scope. Here there should be two input buses so input scope is used.

   `0` – The element to set the property in.

   &busCount – Pointer to the value being set.

   sizeof (busCount) – Size of the value being set.

2. Now increase the maximum frames per slice to use larger slices when the screen is locked.

   ```
   UInt32 maximumFramesPerSlice = 4096;
   result = AudioUnitSetProperty ( mixerUnit,
   kAudioUnitProperty_MaximumFramesPerSlice,
   kAudioUnitScope_Global, 0, &maximumFramesPerSlice,
   sizeof (maximumFramesPerSlice) );
   ```

3. Next attach the input callback to each input bus in the mixer node. The input nodes on the mixer are the only place where audio will enter the processing graph.

   ```
   for (UInt16 busNumber = 0; busNumber < busCount;
        ++busNumber) {

            AURenderCallbackStruct
            inputCallbackStruct;
            inputCallbackStruct.inputProc =
            &inputRenderCallback;
            inputCallbackStruct.inputProcRefCon =
            soundStructArray;
            result = AUGraphSetNodeInputCallback (
            processingGraph, mixerNode, busNumber,
            &inputCallbackStruct );
   }
   ```

   The *inputProcRefCon* variable is what will be passed to the callback—it can be a pointer to anything. Here it is set to *soundStructArray* because that's where the data the callback will need is being kept. *inputRenderCallback* is a static C function which will be defined in *MixerHostAudio.m*.

4. Now set the stream formats for the two input buses.

   ```
   result = AudioUnitSetProperty ( mixerUnit,
   kAudioUnitProperty_StreamFormat,
   kAudioUnitScope_Input, guitarBus,
   &stereoStreamFormat, sizeof
   (stereoStreamFormat) );
   result = AudioUnitSetProperty ( mixerUnit,
   kAudioUnitProperty_StreamFormat,
   kAudioUnitScope_Input, beatsBus,
   &monoStreamFormat, sizeof (monoStreamFormat) );
   ```

5. Finally, set the sample rate of the mixer's output stream.

   ```
   result = AudioUnitSetProperty ( mixerUnit,
   kAudioUnitProperty_SampleRate,
   kAudioUnitScope_Output, 0, &graphSampleRate,
   sizeof (graphSampleRate) );
   ```

Connecting the nodes and initializing the graph

Now that the nodes are good to go, it's possible to connect them and initialize the graph.

```
result = AUGraphConnectNodeInput
( processingGraph, mixerNode, 0, iONode, 0 );
result = AUGraphInitialize (processingGraph);
```

Variable descriptions:
`processingGraph` – The audio processing graph.

`mixerNode` – The mixer unit node, which is the source node.

`0` – The output bus number of the source node. The mixer unit has only one output bus, bus 0.

`iONode` – The I/O unit node, which is the destination node.

`0` – The input bus number of the destination node. The I/O unit has only one input bus, bus 0.

iv. Audio Unit Input Callback Functionality

To provide audio data to the audio processing graph, there is an audio unit input callback function. The graph will call this function every time it needs a buffer of audio, the same way an audio queue callback is used. Here the callback uses data that is stored in the `soundStruct` array rather than reading from a file each time. The callback is a static C function defined within `MixerHostAudio.m`, and its declaration looks like this:

```
static OSStatus inputRenderCallback( void*
inRefCon,AudioUnitRenderActionFlags
*ioActionFlags, const AudioTimeStamp *inTimeStamp,
UInt32 inBusNumber, UInt32 inNumberFrames,
AudioBufferList *ioData )
{
    …
}
```

Variable descriptions:
`void* inRefCon` – A pointer to any struct or object used to manage state. Typically this will be either a custom C struct or an Objective-C or C++ object. In this case, it will be a pointer to the `soundStruct`, which has the necessary data for the callback.

`AudioUnitRenderActionFlags *ioActionFlags` – Unused here. This can be used to mark areas of silence when generating sound.

`const AudioTimeStamp* inTimeStamp` – Unused here.

`UInt32 inBusNumber` – The number of the bus which is calling the callback.

`UInt32 inNumberFrames` – The number of frames being requested.

`AudioBufferList *ioData` – The buffer list which has the buffer to fill.

Getting the soundStruct Array

The first thing the callback should do is get a pointer to the struct or object that has all the state variables. In this case, just cast the `void* inRefCon` as a `soundStructPtr` using a C-style cast:

```
soundStructPtr* soundStructPointerArray =
(soundStructPtr) inRefCon;
```

Also grab some useful data from the sound struct—the total frame count and whether the data is stereo.

```
UInt32 frameTotalForSound =
soundStructPointerArray[inBusNumber].frameCount;
BOOL isStereo = soundStructPointerArray
[inBusNumber].isStereo;
```

Filling a Buffer

Now to fill the buffer with the audio data stored in the `soundStruct` array. First, set up pointers to the input data and the output buffers.

```
AudioUnitSampleType *dataInLeft;
AudioUnitSampleType *dataInRight;
dataInLeft = soundStructPointerArray
[inBusNumber].audioDataLeft;
dataInRight = soundStructPointerArray
[inBusNumber].audioDataRight;
AudioUnitSampleType *outSamplesChannelLeft;
AudioUnitSampleType *outSamplesChannelRight;
outSamplesChannelLeft = (AudioUnitSampleType *)
ioData->mBuffers[0].mData;
outSamplesChannelRight = (AudioUnitSampleType *)
ioData->mBuffer[1].mData;
```

Next, get the sample number to start reading from, which is stored in the `soundStruct` array, and copy the data from the input buffers into the output buffers. When the sample number reaches the end of the file, set it to 0 so that playback will just loop.

```
for (UInt32 frameNumber = 0; frameNumber <
     inNumberFrames; ++frameNumber) {
     outSamplesChannelLeft[frameNumber] =
     dataInLeft [sampleNumber];
     if (isStereo) outSamplesChannelRight
     [frameNumber] = dataInRight[sampleNumber];
          sampleNumber++;

          if (sampleNumber >= frameTotalForSound)
              sampleNumber = 0;
}
soundStructPointerArray[inBusNumber].sampleNumber
= sampleNumber;
```

That's all the callback needs to do—it just fills a buffer whenever it's called. The processing graph will call the callback at the right times to keep the audio playing smoothly.

Processing

After filling the audio buffer, just as in an audio queue callback, it's possible to process the audio directly by manipulating the values in the buffer. And if the application uses synthesis, not playback, the buffer can be filled directly without using a file. This is also where any iZotope effects would be applied. Details regarding the implementation of iZotope effects are distributed along with our effect libraries.

v. Setting Audio Unit Parameters

The Multichannel Mixer audio unit doesn't do anything interesting unless its parameters are changed. That functionality can be put in the *MixerHostAudio* class within a few methods: *enableMixerInput:isOn*:, *setMixerInput:gain*:, and *setMixerOutputGain*:.

Turning Inputs On and Off

To enable and disable the two input buses of the mixer, use `AudioUnitSetParameter()`.

```
OSStatus result = AudioUnitSetParameter
( mixerUnit, kMultiChannelMixerParam_Enable,
kAudioUnitScope_Input, inputBus, isOnValue, 0 );
```

Variable descriptions:
`mixerUnit` – The desired audio unit to set a parameter in.

`kMultiChannelMixerParam_Enable` – The parameter to be set, in this case the enable/disable parameter.

`kAudioUnitScope_Input` – The scope of the parameter. This involves enabling and disabling input buses, so it falls in the mixer's input scope.

`inputBus` – The number of the bus to enable or disable, which is an argument to the method.

`isOnValue` – Whether to enable/disable the bus, also an argument to the method.

`0` – An offset in frames, which Apple recommends always be set to 0.

Then make sure the sample index in both buses matches so that they stay in sync during playback after being disabled.

```
if (0 == inputBuss && 1 == isOnValue) {
   soundStructArray[0].sampleNumber =
soundStructArray[1].sampleNumber;
}
if (1 == inputBus && 1 == isOnValue) {
soundStructArray[1].sampleNumber =
soundStructArray[0].sampleNumber;
}
```

Setting Gain

To set the input gain in the two input buses of the mixer, again use `AudioUnitSetParameter()`. The arguments are identical except for the parameter ID, which is now `kMultiChannelMixerParam_Volume`, and the parameter value to set, which is an argument to this method `newGain`.

```
OSStatus result = AudioUnitSetParameter
( mixerUnit, kMultiChannelMixerParam_Volume,
kAudioUnitScope_Input, inputBus, newGain, 0 );
```

`AudioUnitSetParameter()` is also used to set the output gain of the mixer. The arguments are similar to those for input gain, but the scope is now *kAudioUnitScope_Input* and the element number is 0, because there is only one output element while there are two input buses.

```
OSStatus result = AudioUnitSetParameter
( mixerUnit, kMultiChannelMixerParam_Volume,
kAudioUnitScope_Output, 0, newGain, 0 );
```

vi. Running the Audio Processing Graph

Starting and Stopping

There are also simple interface methods to start and stop the processing graph: `startAUGraph` and `stopAUGraph`. All that needs to be done is call the appropriate functions on the processing graph (and set the *playing* member variable, which is used to handle interruptions). Starting the graph (`startAUGraph`) is done as follows:

```
OSStatus result = AUGraphStart (processingGraph);
self.playing = YES;
```

And to stop it, *stopAUGraph* first checks whether it's running, and stops it if it is.

```
Boolean isRunning = false;
OSStatus result = AUGraphIsRunning
(processingGraph, &isRunning);
    if (isRunning) {
        result = AUGraphStop (processingGraph);
        self.playing = NO;
    }
```

Handling Interruptions

Because it is declared as implementing the <*AVAudioSession Delegate*> protocol, *MixerHostAudio* has methods to handle interruptions to the audio stream, like phone calls and alarms. It is possible to keep track of whether the audio session has been interrupted while playing in a boolean member variable, *interruptedDuringPlayback*. In *beginInterruption*, it's possible to check whether sound is currently playing. If it is, set *interruptedDuringPlayback* and post a notification that the view controller will see.

```
if (playing) {
   self.interruptedDuringPlayback = YES;
   NSString*
   MixerHostAudioObjectPlaybackStateDidChangeNotif
   ication=
   @"MixerHostAudioObjectPlaybackStateDidChangeNot
   ification";
   [[NSNotificationCenter defaultCenter]
   postNotificationName:
   MixerHostAudioObjectPlaybackStateDidChangeNotif
   ication object: self];
}
```

The *MixerHostViewController* has registered to receive these notifications in the *registerForAudioObjectNotifications* method. When it receives such a notification, it will stop playback.

When the audio session resumes from an interruption, *end InterruptionWithFlags*: is called. The *flags* value is a series of bits in which each bit can be a single flag, so it can be checked against *AVAudioSessionInterruptionFlags_ShouldResume* with a bitwise and, &.

```
if (flags &
AVAudioSessionInterruptionFlags_ShouldResume) {…}
```

If the session should resume, reactivate the audio session:

```
[[AVAudioSession sharedInstance] setActive: YES
error: &endInterruptionError];
```

Then update the *interruptedDuringPlayback* value and post a notification to resume playback if necessary.

```
if (interruptedDuringPlayback) {
   self.interruptedDuringPlayback = NO;
   NSString*
   MixerHostAudioObjectPlaybackStateDidChangeNotif
   ication =
   @"MixerHostAudioObjectPlaybackStateDidChangeNot
   ification";
   [[NSNotificationCenter defaultCenter]
   PostNotificationName:
   MixerHostAudioObjectPlaybackStateDidChangeNotif
   ication object: self];
}
```

vii. aurioTouch

The **aurioTouch** sample application also uses audio units, but in a different way. First of all, it uses the Remote I/O unit for input as well as output, and this is the only unit that is used. (The aurioTouch project is also not quite as neat and well-commented as MixerHost, so it may be a little harder to understand). This document will cover the structure of how aurioTouch works but won’t dive too deep into its complexities.

Structure

Most of the audio functionality of aurioTouch is in *aurioTouchApp Delegate.mm*. This is where an interruption listener *rio InterruptionListener()*, a property listener *propListener*(), and an audio unit callback *PerformThru()* can be found. Most audio and graphics initialization takes place in *applicationDidFinish Launching*:, while setup for the Remote I/O unit takes place in *aurio_helper.cpp*, in the *SetupRemoteIO* function. Most of the other code in the app is for its fast Fourier transform (FFT) functionality, and for graphics.

Initializing the Audio Session

In aurioTouch, *applicationDidFinishLaunching*: initializes an audio session similar to the way MixerHost does. First, the callback is stored in an *AURenderCallback* struct *inputProc* for use with the I/O unit. *inputProcRefCon* is a pointer to desired data to retrieve when the callback is called, so here it's just a reference to the *aurioTouchAppDelegate* object. (In the three examples in this document, its been used as a reference to a C++ object, a C struct, and an Objective-C object.)

```
inputProc.inputProc = PerformThru;
inputProc.inputProcRefCon = self;
```

Then it initializes a *SystemSoundID* named *buttonPressSound*, which is functionality which hasn't been seen before. Through Audio Services, it's possible to play simple sounds without handling the audio directly at the sample level (again the error-checking code is left out for clarity here).

```
url = CFURLCreateWithFileSystemPath
( kCFAllocatorDefault, CFStringRef([[NSBundle
mainBundle] pathForResource: @"button_press"
ofType:@"caf"]), kCFURLPOSIXPathStyle, false);
AudioServicesCreateSystemSoundID(url,
&buttonPressSound);
CFRelease(url);
```

Now it initializes the audio session, sets some desired properties, and activates it. Unlike MixerHost, the session category is play and record because input audio is being used. Just as was done in MixerHost, there is a request for a preferred buffer size, but the actual buffer size must be retrieved because there's no guarantee they match.

```
AudioSessionInitialize(NULL, NULL,
rioInterruptionListener, self);
UInt32 audioCategory =
kAudioSessionCategory_PlayAndRecord;
AudioSessionSetProperty
( kAudioSessionProperty_AudioCategory, sizeof
(audioCategory), &audioCategory);
AudioSessionAddPropertyListener
( kAudioSessionProperty_AudioRouteChange,
propListener, self);
Float32 preferredBufferSize = .005;
AudioSessionSetProperty
(
kAudioSessionProperty_PreferredHardwareIOBufferDur
ation, sizeof(preferredBufferSize),
&preferredBufferSize);
UInt32 size = sizeof(hwSampleRate);
AudioSessionGetProperty
( kAudioSessionProperty_CurrentHardwareSampleRate,
&size, &hwSampleRate);
AudioSessionSetActive(true);
```

Next it initializes the Remote I/O unit using `SetupRemoteIO()`, which will be covered in the next section. Hand it the audio unit, the callback, and the audio data format.

```
SetupRemoteIO(rioUnit, inputProc, thruFormat);
```

Then aurioTouch sets up custom objects for DC filtering and FFT processing, which won't be examined in detail here. Next it starts the Remote I/O unit and sets thruFormat to match the audio unit's output format.

```
AudioOutputUnitStart(rioUnit);
AudioUnitGetProperty(rioUnit,
kAudioUnitProperty_StreamFormat,
kAudioUnitScope_Output, 1, &thruFormat, &size);
```

Initializing the Remote I/O Unit

Set up the Remote I/O unit in `SetupRemoteIO()` in *aurio_helper.cpp*. This is similar to what was done in MixerHost. First create a description of the I/O unit component.

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

Because there is only one audio unit in aurioTouch, there is no audio processing graph. So instead of adding a node to a graph with this description, just explicitly search for a component that matches this description and create a new instance of it. The only component that will match this description is the Remote I/O unit.

```
AudioComponent comp = AudioComponentFindNext(NULL,
&desc);
AudioComponentInstanceNew(comp, &inRemoteIOUnit);
```

Now enable input from the I/O unit so that it will provide audio, and then register the callback as the render callback.

```
UInt32 one = 1;
AudioUnitSetProperty(inRemoteIOUnit,
kAudioOutputUnitProperty_EnableIO,
kAudioUnitScope_Input, 1, &one, sizeof(one));
AudioUnitSetProperty(inRemoteIOUnit,
kAudioUnitProperty_SetRenderCallback,
kAudioUnitScope_Input, 0, &inRenderProc, sizeof
(inRenderProc));
```

Finally set the format for both input and output and initialize the unit. `SetAUCanonical()` is a function from `CAStreamBasicDescription.h` which creates a canonical AU format with the given number of channels. *false* means it won't be interleaved.

```
outFormat.SetAUCanonical(2, false);
outFormat.mSampleRate = 44100;
AudioUnitSetProperty(inRemoteIOUnit,
kAudioUnitProperty_StreamFormat,
kAudioUnitScope_Input, 0, &outFormat, sizeof
(outFormat));
AudioUnitSetProperty(inRemoteIOUnit,
kAudioUnitProperty_StreamFormat,
kAudioUnitScope_Output, 1, &outFormat, sizeof
(outFormat));
AudioUnitInitialize(inRemoteIOUnit);
```

The Audio Unit Render Callback

Take a look at the render callback `PerformThru` in `aurioTouchAppDelegate.mm`. Most of the code here is for preparing the data so that its values can be shown in an oscilloscope or FFT display, so that won't be covered in great detail. Notice that the first thing done in the callback is to actually render the audio using the Remote I/O unit:

```
aurioTouchAppDelegate *THIS =
(aurioTouchAppDelegate *)inRefCon;
OSStatus err = AudioUnitRender(THIS->rioUnit,
ioActionFlags, inTimeStamp, 1, inNumberFrames,
ioData);
```

Next the buffer is processed using a DC filter. A DC filter removes the DC component of an audio signal, which is, roughly speaking, the part of the signal that is a constant positive or negative voltage. In other words, it normalizes the signal so that its mean value is 0. This is an example of how input audio can be processed in a callback as well as audio for playback.

```
for(UInt32 i = 0; i < ioData->mNumberBuffers; ++i)
   THIS->dcFilter[i].InplaceFilter((SInt32*)
   (ioData->mBuffers[i].mData), inNumberFrames,
   1);
```

Then pass the buffer of audio to drawing functions depending on which drawing mode is enabled. These functions will not alter the audio data, just read it to draw in the oscilloscope display or the FFT.

```
if (THIS->displayMode ==
aurioTouchDisplayModeOscilloscopeWaveform)
{
    …
}
else if ((THIS->displayMode ==
aurioTouchDisplayModeSpectrum) || (THIS-
>displayMode ==
aurioTouchDisplayModeOscilloscopeFFT))
    {
        …
    }
```

Finally, check whether the `mute` member variable is on. If it is, silence the data in the buffer using the `SilenceData()` function, which just zeroes out the buffers. The buffer is passed through the Remote I/O unit and is played, whether it is silence or normal audio.

```
if (THIS->mute == YES) { SilenceData(ioData); }
```

3. Glossary

ADC – An analog to digital converter, a device that will sample an analog sound at a fixed sampling rate and convert it into a digital signal.

amplitude – The strength of vibration of a sound wave—roughly speaking, the ‘volume’ of a sound at a particular moment.

analog – A continuous ‘real-world’ signal rather than discrete numbers. An analog signal must be sampled through an ADC in order to be converted to a digital signal that may be manipulated through processing in a computer.

audio queue – A Core Audio service for working with audio at a high level. Consists of a short list of buffers that are passed to the hardware and then reused. The audio queue object will call the callback whenever it needs a buffer filled.

audio unit – A Core Audio service for working with audio at a low level. An individual audio unit is a component, like a mixer, an equalizer, or an I/O unit, which takes audio input and produces audio output. Audio units are ‘plugged’ into one another, creating a chain of audio processing components that provide real-time audio processing.

bit – A binary digit. The smallest computational unit provided by a computer. A bit has only two states, on or off (1 or 0) and represents a single piece of binary data.

bit depth – The total possible size in bits of each sample in a signal—for example, an individual sample might consist of a signed 16-bit integer, where 32,767 (2^16 - 1) is the maximum amplitude, and -32,768 is the minimum.

buffer – A short list of samples whose length is convenient for processing. For live audio processing, buffers are typically small, a few hundred to a few thousand samples (usually in powers of two), in order to minimize latency. For offline processing, buffers may be larger. Buffers are time efficient because processing functions are only called every time a buffer is needed, and they’re space efficient because they don’t occupy a huge amount of memory. Typical buffer sizes are 256, 512, 1024 and 2048 samples, though they may vary depending on the application.

callback – A written function that the system calls. When working with audio, callbacks are used to provide buffers whenever they are requested. Callbacks for audio input will have a few arguments, including a pointer to a buffer that should be filled and an object or struct containing additional information. The callback is where an audio buffer is handed off to an iZotope effect for processing, and upon return that audio is passed to the audio hardware for output

channel – An individual audio signal in a stream. Channels are independent of one another. There may be one or many channels in an audio stream or file—for example, a mono sound has one channel and a stereo sound has two channels.

channel count – The number of individual channels in an input stream.

Core Audio – The general API for audio in Mac OS and iOS, which includes audio queues and audio units along with a large base of other functionality for working with audio.

DAC – A digital to analog converter. A device which reconstructs a continuous signal from individual samples by interpolating what should be between the samples.

deinterleaved – See **interleaved**.

digital – Consisting of discrete numbers rather than a continuous signal.

fixed-point – A system for representing numbers that are essentially integers. This data type has a fixed number of digits after the decimal point (radix point). A typical fixed point number may look like 12345.

floating-point – A system for representing numbers that include a decimal place. Floating point refers to the fact that the decimal place can be placed anywhere relative to the significant digits of a number. Floating point numbers can typically represent a larger range of values than their fixed point counterparts at the expense of additional memory. A floating point number may look like 123.45 or 1.2345.

frame – A set of samples that occur at the same time. For example, stereo audio will have two samples in each frame.

L	**R**	**L**	**R**	**L**	**R**	**L**	**R**	...

Frame	Frame	Frame	Frame	...

Buffer or packet	...

frequency - How often an event occurs. For sounds, this refers to the frequency of oscillation of the sound wave. Frequency is measured in Hertz.

Hertz or **Hz** - A unit of frequency—cycles per second.

interleaved and **deinterleaved** – These terms refer to the way samples are organized by channel in audio data. **Interleaved** audio data is data in which samples are stored in a single dimensional array. It's called interleaved because the audio alternates between channels, providing the first sample of the first channel followed by the first sample of the second channel and so on. So, where **L** is a sample from the left channel and **R** is a sample from the right, it's organized as follows:

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	...
L	**R**	**L**	**R**	**L**	**R**	**L**	**R**	**L**	**R**	**L**	**R**	**L**	**R**	**L**	**R**	...

In contrast, a **deinterleaved** or **noninterleaved** audio stream in which the channels are separated from one another and each channel is represented as its own contiguous group of samples. A deinterleaved audio stream may be stored in a two-dimensional array with each dimension consisting of a single channel.

0	1	2	3	4	5	6	7	...
L	**L**	**L**	**L**	**L**	**L**	**L**	**L**	...
R	**R**	**R**	**R**	**R**	**R**	**R**	**R**	...

latency - Time delay due to processing. This refers to the amount of time that passes when audio is sent to a processing function and when that audio is output. Any audio processing system will have some amount of inherent delay due to the amount of time it takes to process the audio. Minimizing the amount of latency is desirable for real-time audio application.

noninterleaved – See **interleaved**.

packet – A small chunk of audio that is read from a file.

sample - A single number indicating the amplitude of a sound at a particular moment. Digital audio is represented as a series of samples at some sampling rate, such as 44,100 or 48,000 Hz, and with some bit depth, such as 16 bits.

sampling rate - The frequency with which samples of a sound are taken. Typical values are 22,050, 44,100 or 48,000 samples per second, or Hz. Your trusty old compact disc player hums along at a sampling rate of 44.1 kHz, or 44,100 Hz.

4. Additional Help and Resources

The Apple Developer documentation, sample applications and forums are great places to find additional information on how to develop audio applications for iOS. Here are some useful links:

Audio Queues

The Audio Queue Services Programming Guide from the Mac OS X Developer Library: http://developer.apple.com/library/mac/#documentation/MusicAudio/Conceptual/AudioQueueProgrammingGuide/Introduction/Introduction.html

The Audio Queue Services Reference from the Mac OS X Developer Library: http://developer.apple.com/library/mac/#documentation/MusicAudio/Reference/AudioQueueReference/Reference/reference.html

The SpeakHere sample application: http://developer.apple.com/library/ios/#samplecode/SpeakHere/Introduction/Intro.html

Audio Units

The Audio Unit Hosting Guide for iOS from the iOS Developer Library: http://developer.apple.com/library/ios/#documentation/MusicAudio/Conceptual/AudioUnitHostingGuide_iOS/Introduction/Introduction.html

The MixerHost sample application: http://developer.apple.com/library/ios/#samplecode/MixerHost/Introduction/Intro.html

The aurioTouch sample application: http://developer.apple.com/library/ios/#samplecode/aurioTouch/Introduction/Intro.html

The iPhoneMixerEQGraphTest sample application: http://developer.apple.com/library/ios/#samplecode/iPhoneMixerEQGraphTest/Introduction/Intro.html

General iOS Help

The iOS Developer Library: http://developer.apple.com/library/ios/navigation/

The Apple Developer Forums: http://developer.apple.com/devforums/

5. Appendix - Introductory DSP Tutorial

This appendix is meant to serve as a brief introduction to digital signal processing for beginners. To work with audio in iOS, the reader should be familiar with the ideas described here.

Sound is vibrating air, and at its simplest a sound is just a single sine wave. A sine wave has a **frequency** (the number of times it oscillates per second) and an **amplitude** (the strength of the vibration). The frequency of a sound determines its perceived pitch (whether it is high or low), and the amplitude determines its perceived volume (whether it is loud or soft.) Typically we measure frequency in oscillations per second, or **Hertz (Hz)**. Amplitude can be measured in a few ways; however, the typical unit is **decibels** or **dB**. Decibels are on a logarithmic scale that generally replicates how we perceive the loudness of a signal. In theory, any sound can be represented as a sum of sine waves with different frequencies and amplitudes.

The strength of the air vibration caused by a sound is called the amplitude, and may be thought of as the volume. Sounds heard in this fashion are analog—continuous in both frequency and amplitude. Analog sound that is recorded through a microphone is turned into an electrical signal whose voltage varies with the amplitude of the sound. This voltage is then sampled very rapidly by an analog to digital converter, or **ADC**, at some **sampling rate**, with typical values being 44,100 and 48,000 samples per second, or Hz.

The numbers obtained from this process are then stored in a digital device as a series of **samples**. The signal is **digital** because those samples are discrete and are represented as numbers with a set amount of digits. The possible size of those numbers in bits is called the **bit depth**—for example, samples might be **fixed-point** signed 16-bit integers, where 32,767 (2^16 - 1) is the maximum amplitude, and -32,768 is the minimum. Or samples may be **floating-point** values, normally on a scale from -1.0 to 1.0. The amplitude of a digital signal may also be expressed in **decibels** (dB), but unlike the dB values for real-world sounds, the values are placed on a scale where 0.0 dB is the maximum amplitude a system can produce. Thus the representation of decibels in digital systems is in the negative range. The typical person can hear a signal between 0.0dB (very loud) and -60.0dB (very quiet) from a digital system.

Once the signal is digital, a computer can manipulate it in a variety of ways through **digital signal processing**. DSP is simply the mathematics used for effects, noise reduction, metering, and a wide array of other audio applications.

After the audio is processed through DSP algorithms, it must be converted to sound through a digital to analog converter, or **DAC.** The DAC will reconstruct a continuous waveform by **interpolating** the values it expects should be between the samples and playing that audio back through speakers or headphones.