Audio Queue Services Programming Guide

Developer

Contents

Figures and Listings

Introduction

This document describes how to use Audio Queue Services, a C programming interface in Core Audio's Audio Toolbox framework.

What Is Audio Queue Services?

Audio Queue Services provides a straightforward, low overhead way to record and play audio in iOS and Mac OS X. It is the recommended technology to use for adding basic recording or playback features to your iOS or Mac OS X application.

Audio Queue Services lets you record and play audio in any of the following formats:

- Linear PCM.
- Any compressed format supported natively on the Apple platform you are developing for.
- Any other format for which a user has an installed codec.

Audio Queue Services is high level. It lets your application use hardware recording and playback devices (such as microphones and loudspeakers) without knowledge of the hardware interface. It also lets you use sophisticated codecs without knowledge of how the codecs work.

At the same time, Audio Queue Services supports some advanced features. It provides fine-grained timing control to support scheduled playback and synchronization. You can use it to synchronize playback of multiple audio queues and to synchronize audio with video.

> **Note:** Audio Queue Services provides features similar to those previously provided by the Sound Manager in Mac OS X. It adds additional features such as synchronization. The Sound Manager is deprecated in Mac OS X v10.5 and does not work with 64-bit applications. Apple recommends Audio Queue Services for all new development and as a replacement for the Sound Manager in existing Mac OS X applications.

Audio Queue Services is a pure C interface that you can use in Cocoa applications as well as in Mac OS X command-line tools. To help keep the focus on Audio Queue Services, the code examples in this document are sometimes simplified by using C++ classes from the Core Audio SDK. However, neither the SDK nor the C++ language is necessary to use Audio Queue Services.

Who Should Read This Guide?

Audio Queue Services Programming Guide is useful to all iOS and Mac OS X developers who want a streamlined, straightforward way to record or play audio. To get the most from this document, you should be familiar with:

- The C programming language
- Using Xcode to build iOS or Mac OS X applications
- The terminology described in *Core Audio Glossary*

Organization of This Document

This guide contains the following chapters:

- About Audio Queues (page 9) describes the capabilities, architecture, and internal workings of audio queues.
- Recording Audio (page 23) describes how to record audio.
- Playing Audio (page 41) describes how to play audio.

See Also

You may find the following documents helpful:

- The companion document *Audio Queue Services Reference* provides descriptions of the functions, callbacks, constants, and data types in Audio Queue Services.
- *Core Audio Data Types Reference* describes data types essential for using Audio Queue Services.

- *Core Audio Overview* provides a summary of the Core Audio frameworks, and includes an appendix on Supported Audio File and Data Formats in OS X.
- *Core Audio Glossary* defines key terms used in the Core Audio documentation.

About Audio Queues

In this chapter you learn about the capabilities, architecture, and internal workings of audio queues. You get introduced to audio queues, audio queue buffers, and the callback functions that audio queues use for recording or playback. You also find out about audio queue states and parameters. By the end of this chapter you will have gained the conceptual understanding you need to use this technology effectively.

What Is an Audio Queue?

An **audio queue** is a software object you use for recording or playing audio in iOS or Mac OS X. It is represented by the `AudioQueueRef` opaque data type, declared in the `AudioQueue.h` header file.

An audio queue does the work of:

- Connecting to audio hardware
- Managing memory
- Employing codecs, as needed, for compressed audio formats
- Mediating recording or playback

You can use audio queues with other Core Audio interfaces, and a relatively small amount of custom code, to create a complete digital audio recording or playback solution in your application.

Audio Queue Architecture

All audio queues have the same general structure, consisting of these parts:

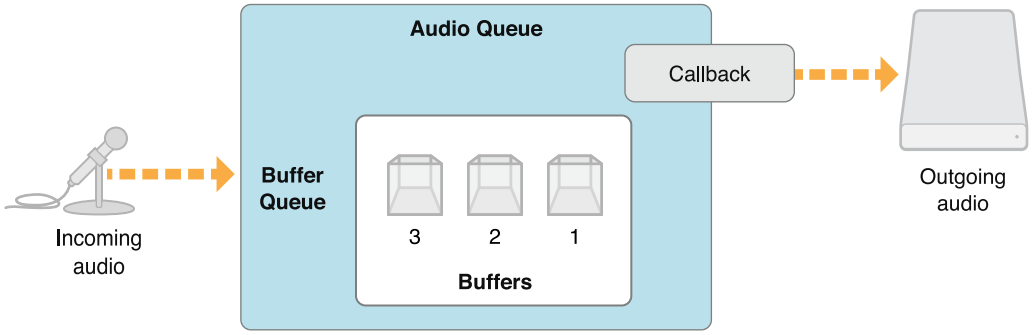
- A set of **audio queue buffers**, each of which is a temporary repository for some audio data
- A **buffer queue**, an ordered list for the audio queue buffers
- An **audio queue callback** function, that you write

The architecture varies depending on whether an audio queue is for recording or playback. The differences are in how the audio queue connects its input and output, and in the role of the callback function.

Audio Queues for Recording

A recording audio queue, created with the `AudioQueueNewInput` function, has the structure shown in Figure 1-1.

Figure 1-1 A recording audio queue



The input side of a recording audio queue typically connects to external audio hardware, such as a microphone. In iOS, the audio comes from the device connected by the user—built-in microphone or headset microphone, for example. In the default case for Mac OS X, the audio comes from the system's default audio input device as set by a user in System Preferences.

The output side of a recording audio queue makes use of a callback function that you write. When recording to disk, the callback writes buffers of new audio data, that it receives from its audio queue, to an audio file. However, recording audio queues can be used in other ways. You could also use one, for example, in a realtime audio analyzer. In such a case, your callback would provide audio data directly to your application instead of writing it to disk.

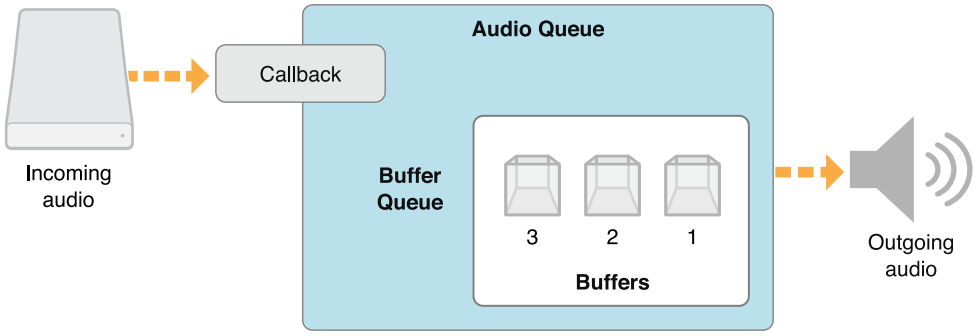
You'll learn more about this callback in The Recording Audio Queue Callback Function (page 17).

Every audio queue—whether for recording or playback—has one or more audio queue buffers. These buffers are arranged in a specific sequence called a buffer queue. In the figure, the audio queue buffers are numbered according to the order in which they are filled—which is the same order in which they are handed off to the callback. You'll learn how an audio queue uses its buffers in The Buffer Queue and Enqueuing (page 12).

Audio Queues for Playback

A playback audio queue (created with the `AudioQueueNewOutput` function) has the structure shown in Figure 1-2.

Figure 1-2 A playback audio queue



In a playback audio queue, the callback is on the input side. The callback is responsible for obtaining audio data from disk (or some other source) and handing it off to the audio queue. Playback callbacks also tell their audio queues to stop when there's no more data to play. You'll learn more about this callback in The Playback Audio Queue Callback Function (page 18).

A playback audio queue's output typically connects to external audio hardware, such as a loudspeaker. In iOS, the audio goes to the device chosen by the user—for example, the receiver or the headset. In the default case in Mac OS X, the audio goes to the system's default audio output device as set by a user in System Preferences.

Audio Queue Buffers

An **audio queue buffer** is a data structure, of type `AudioQueueBuffer`, as declared in the `AudioQueue.h` header file:

```
typedef struct AudioQueueBuffer {
    const UInt32   mAudioDataBytesCapacity;
    void *const    mAudioData;
    UInt32         mAudioDataByteSize;
    void           *mUserData;
} AudioQueueBuffer;
typedef AudioQueueBuffer *AudioQueueBufferRef;
```

The `mAudioData` field, highlighted in the code listing, points to the buffer per se: a block of memory that serves as a container for transient blocks of audio data being played or recorded. The information in the other fields helps an audio queue manage the buffer.

An audio queue can use any number of buffers. Your application specifies how many. A typical number is three. This allows one to be busy with, say, writing to disk while another is being filled with fresh audio data. The third buffer is available if needed to compensate for such things as disk I/O delays. Figure 1-3 (page 13) illustrates this.

Audio queues perform memory management for their buffers.

- An audio queue allocates a buffer when you call the `AudioQueueAllocateBuffer` function.
- When you release an audio queue by calling the `AudioQueueDispose` function, the queue releases its buffers.

This improves the robustness of the recording and playback features you add to your application. It also helps optimize resource usage.

For a complete description of the `AudioQueueBuffer` data structure, see *Audio Queue Services Reference*.

The Buffer Queue and Enqueuing

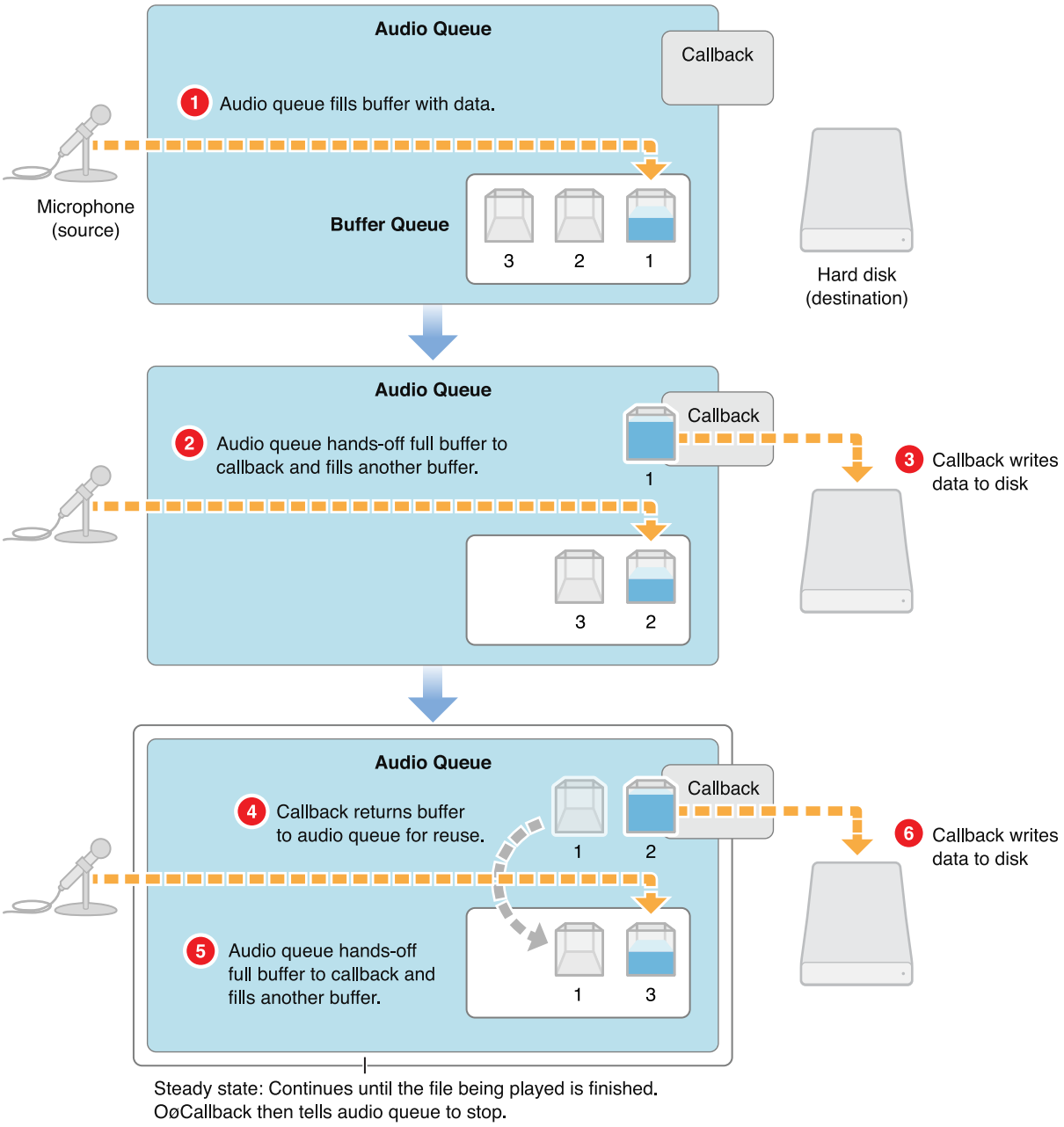
The buffer queue is what gives audio queues, and indeed Audio Queue Services, their names. You met the buffer queue—an ordered list of buffers—in Audio Queue Architecture (page 9). Here you learn about how an audio queue object, together with your callback function, manage the buffer queue during recording or playback. In particular, you learn about **enqueuing**, the addition of an audio queue buffer to a buffer queue. Whether you are implementing recording or playback, enqueuing is a task that your callback performs.

The Recording Process

When recording, one audio queue buffer is being filled with audio data acquired from an input device, such as a microphone. The remaining buffers in the buffer queue are lined up behind the current buffer, waiting to be filled with audio data in turn.

The audio queue hands off filled buffers of audio data to your callback in the order in which they were acquired. Figure 1-3 illustrates how recording works when using an audio queue.

Figure 1-3 The recording process



In step 1 of Figure 1-3, recording begins. The audio queue fills a buffer with acquired data.

In step 2, the first buffer has been filled. The audio queue invokes the callback, handing it the full buffer (buffer 1). The callback (step 3) writes the contents of the buffer to an audio file. At the same time, the audio queue fills another buffer (buffer 2) with freshly acquired data.

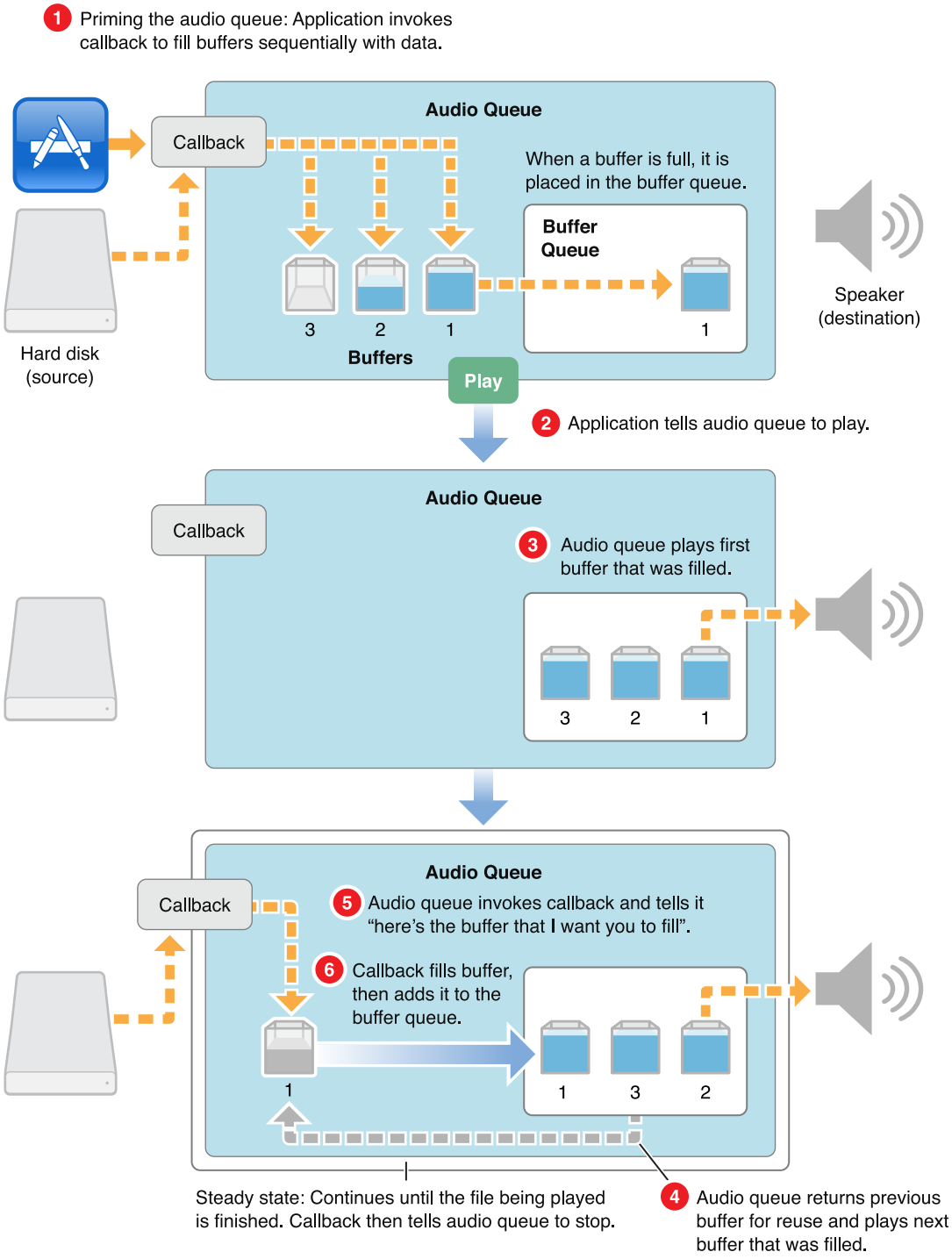
In step 4, the callback enqueues the buffer (buffer 1) that it has just written to disk, putting it in line to be filled again. The audio queue again invokes the callback (step 5), handing it the next full buffer (buffer 2). The callback (step 6) writes the contents of this buffer to the audio file. This looping steady state continues until the user stops the recording.

The Playback Process

When playing, one audio queue buffer is being sent to an output device, such as a loudspeaker. The remaining buffers in the buffer queue are lined up behind the current buffer, waiting to be played in turn.

The audio queue hands off played buffers of audio data to your callback in the order in which they were played. The callback reads new audio data into a buffer and then enqueues it. Figure 1-4 illustrates how playback works when using an audio queue.

Figure 1-4 The playback process



In step 1 of Figure 1-4, the application primes the playback audio queue. The application invokes the callback once for each of the audio queue buffers, filling them and adding them to the buffer queue. Priming ensures that playback can start instantly when your application calls the `AudioQueueStart` function (step 2).

In step 3, the audio queue sends the first buffer (buffer 1) to output.

As soon as the first buffer has been played, the playback audio queue enters a looping steady state. The audio queue starts playing the next buffer (buffer 2, step 4) and invokes the callback (step 5), handing it the just-played buffer (buffer 1). The callback (step 6) fills the buffer from the audio file and then enqueues it for playback.

Controlling the Playback Process

Audio queue buffers are always played in the order in which they are enqueued. However, Audio Queue Services provides you with some control over the playback process with the`AudioQueueEnqueueBufferWithParameters` function. This function lets you:

- Set the precise playback time for a buffer. This lets you support synchronization.
- Trim frames at the start or end of an audio queue buffer. This lets you remove leading or trailing silence.
- Set the playback gain at the granularity of a buffer.

For more about setting playback gain, see Audio Queue Parameters (page 21). For a complete description of the `AudioQueueEnqueueBufferWithParameters` function, see *Audio Queue Services Reference*.

The Audio Queue Callback Function

Typically, the bulk of your programming work in using Audio Queue Services consists of writing an audio queue callback function.

During recording or playback, an audio queue callback is invoked repeatedly by the audio queue that owns it. The time between calls depends on the capacity of the audio queue's buffers and will typically range from half a second to several seconds.

One responsibility of an audio queue callback, whether it is for recording or playback, is to return audio queue buffers to the buffer queue. The callback adds a buffer to the end of the buffer queue using the `AudioQueueEnqueueBuffer` function. For playback, you can instead use the `AudioQueueEnqueueBufferWithParameters` function if you need more control, as described in Controlling the Playback Process (page 16).

The Recording Audio Queue Callback Function

This section introduces the callback you'd write for the common case of recording audio to an on-disk file. Here is the prototype for a recording audio queue callback, as declared in the `AudioQueue.h` header file:

```
AudioQueueInputCallback (
    void                                *inUserData,
    AudioQueueRef                       inAQ,
    AudioQueueBufferRef                 inBuffer,
    const AudioTimeStamp                *inStartTime,
    UInt32                              inNumberPacketDescriptions,
    const AudioStreamPacketDescription *inPacketDescs
);
```

A recording audio queue, in invoking your callback, supplies everything the callback needs to write the next set of audio data to the audio file:

- `inUserData` is, typically, a custom structure that you've set up to contain state information for the audio queue and its buffers, an audio file object (of type `AudioFileID`) representing the file you're writing to, and audio data format information for the file.
- `inAQ` is the audio queue that invoked the callback.
- `inBuffer` is an audio queue buffer, freshly filled by the audio queue, containing the new data your callback needs to write to disk. The data is already formatted according to the format you specify in the custom structure (passed in the `inUserData` parameter). For more on this, see Using Codecs and Audio Data Formats (page 18).
- `inStartTime` is the sample time of the first sample in the buffer. For basic recording, your callback doesn't use this parameter.
- `inNumberPacketDescriptions` is the number of packet descriptions in the `inPacketDescs` parameter. If you are recording to a VBR (variable bitrate) format, the audio queue supplies a value for this parameter to your callback, which in turn passes it on to the `AudioFileWritePackets` function. CBR (constant bitrate) formats don't use packet descriptions. For a CBR recording, the audio queue sets this and the `inPacketDescs` parameter to `NULL`.
- `inPacketDescs` is the set of packet descriptions corresponding to the samples in the buffer. Again, the audio queue supplies the value for this parameter, if the audio data is in a VBR format, and your callback passes it on to the `AudioFileWritePackets` function (declared in the `AudioFile.h` header file).

For more information on the recording callback, see Recording Audio (page 23) in this document, and see *Audio Queue Services Reference*.

The Playback Audio Queue Callback Function

This section introduces the callback you'd write for the common case of playing audio from an on-disk file. Here is the prototype for a playback audio queue callback, as declared in the `AudioQueue.h` header file:

```
AudioQueueOutputCallback (
    void                 *inUserData,
    AudioQueueRef        inAQ,
    AudioQueueBufferRef  inBuffer
);
```

A playback audio queue, in invoking your callback, supplies what the callback needs to read the next set of audio data from the audio file:

- `inUserData` is, typically, a custom structure that you've set up to contain state information for the audio queue and its buffers, an audio file object (of type `AudioFileID`) representing the file you're writing to, and audio data format information for the file.

  In the case of a playback audio queue, your callback keeps track of the current packet index using a field in this structure.
- `inAQ` is the audio queue that invoked the callback.
- `inBuffer` is an audio queue buffer, made available by the audio queue, that your callback is to fill with the next set of data read from the file being played.

If your application is playing back VBR data, the callback needs to get the packet information for the audio data it's reading. It does this by calling the `AudioFileReadPackets` function, declared in the `AudioFile.h` header file. The callback then places the packet information in the custom data structure to make it available to the playback audio queue.

For more information on the playback callback, see Playing Audio (page 41) in this document, and see *Audio Queue Services Reference*.

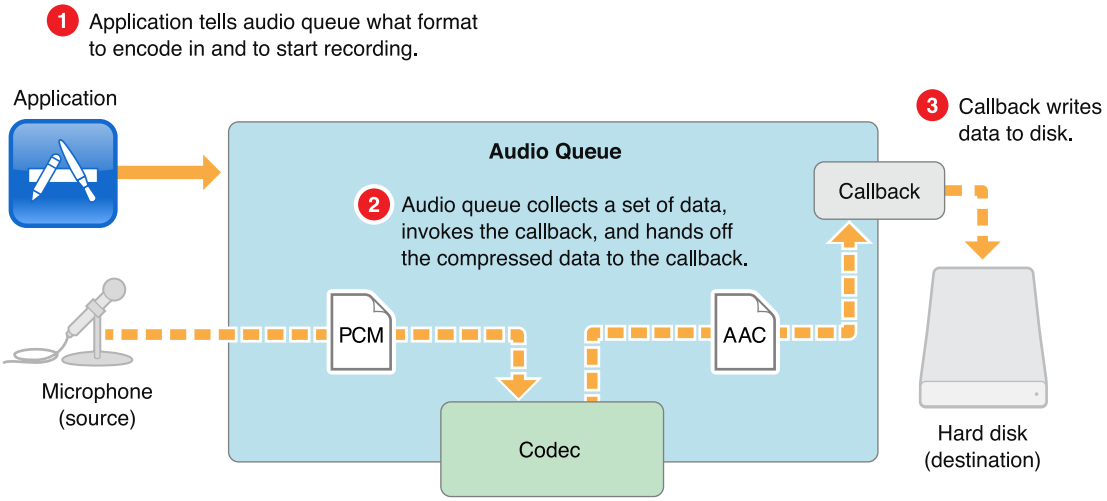
Using Codecs and Audio Data Formats

Audio Queue Services employs codecs (audio data coding/decoding components) as needed for converting between audio formats. Your recording or playback application can use any audio format for which there is an installed codec. You do not need to write custom code to handle various audio formats. Specifically, your callback does not need to know about data formats.

Here's how this works. Each audio queue has an audio data format, represented in an `AudioStreamBasicDescription` structure. When you specify the format—in the `mFormatID` field of the structure—the audio queue uses the appropriate codec. You then specify sample rate and channel count, and that's all there is to it. You'll see examples of setting audio data format in Recording Audio (page 23) and Playing Audio (page 41).

A recording audio queue makes use of an installed codec as shown in Figure 1-5.

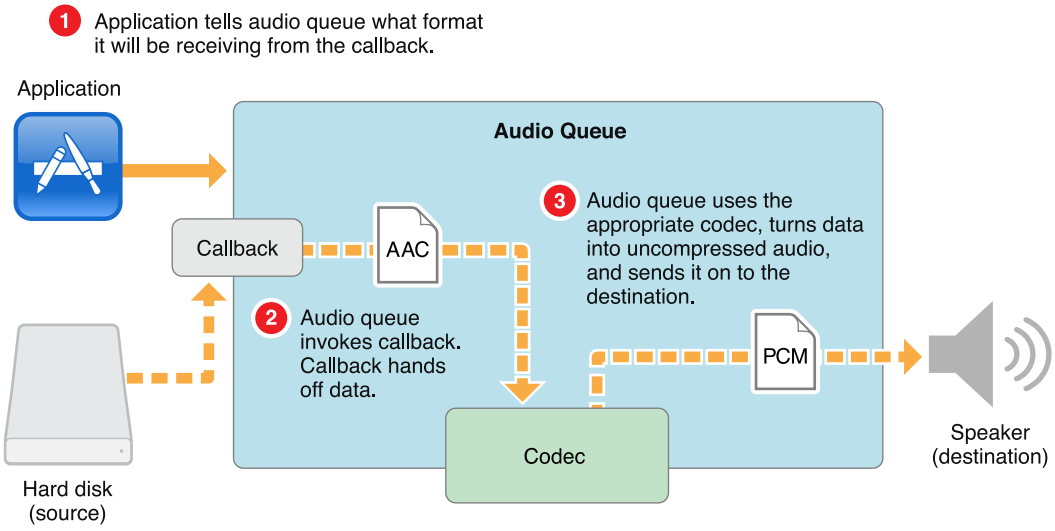
Figure 1-5 Audio format conversion during recording



In step 1 of Figure 1-5, your application tells an audio queue to start recording, and also tells it the data format to use. In step 2, the audio queue obtains new audio data and converts it, using a codec, according to the format you've specified. The audio queue then invokes the callback, handing it a buffer containing appropriately formated audio data. In step 3, your callback writes the formatted audio data to disk. Again, your callback does not need to know about the data formats.

A playback audio queue makes use of an installed codec as shown in Figure 1-6.

Figure 1-6 Audio format conversion during playback



In step 1 of Figure 1-6, your application tells an audio queue to start playing, and also tells it the data format contained in the audio file to be played. In step 2, the audio queue invokes your callback, which reads data from the audio file. The callback hands off the data, in its original format, to the audio queue. In step 3, the audio queue uses the appropriate codec and then sends the audio along to the destination.

An audio queue can make use of any installed codec, whether native to Mac OS X or provided by a third party. To designate a codec to use, you supply its four-character code ID to an audio queue's `AudioStreamBasicDescription` structure. You'll see an example of this in Recording Audio (page 23).

Mac OS X includes a wide range of audio codecs, as listed in the format IDs enumeration in the `CoreAudioTypes.h` header file and as documented in *Core Audio Data Types Reference*. You can determine the codecs available on a system by using the interfaces in the `AudioFormat.h` header file, in the Audio Toolbox Framework. You can display the codecs on a system using the Fiendishthngs application, available as sample code at http://developer.apple.com/samplecode/Fiendishthngs/.

Audio Queue Control and State

An audio queue has a life cycle between creation and disposal. Your application manages this life cycle—and controls the audio queue's state—using six functions declared in the `AudioQueue.h` header file:

- **Start** (`AudioQueueStart`). Call to initiate recording or playback.

- **Prime** (`AudioQueuePrime`). For playback, call before calling `AudioQueueStart` to ensure that there is data available immediately for the audio queue to play. This function is not relevant to recording.
- **Stop** (`AudioQueueStop`). Call to reset the audio queue (see the description below for `AudioQueueReset`) and to then stop recording or playback. A playback audio queue callback calls this function when there's no more data to play.
- **Pause** (`AudioQueuePause`). Call to pause recording or playback without affecting buffers or resetting the audio queue. To resume, call the `AudioQueueStart` function.
- **Flush** (`AudioQueueFlush`). Call after enqueuing the last audio queue buffer to ensure that all buffered data, as well as all audio data in the midst of processing, gets recorded or played.
- **Reset** (`AudioQueueReset`). Call to immediately silence an audio queue, remove all buffers from previously scheduled use, and reset all decoder and DSP state.

You can use the `AudioQueueStop` function in a synchronous or asynchronous mode:

- **Synchronous** stopping happens immediately, without regard for previously buffered audio data.
- **Asynchronous** stopping happens after all queued buffers have been played or recorded.

See *Audio Queue Services Reference* for a complete description of each of these functions, including more information on synchronous and asynchronous stopping of audio queues.

Audio Queue Parameters

An audio queue has adjustable settings called **parameters**. Each parameter has an enumeration constant as its key, and a floating-point number as its value. Parameters are typically used in playback, not recording.

In Mac OS X v10.5, the only audio queue parameter available is for gain. The value for this parameter is set or retrieved using the `kAudioQueueParam_Volume` constant, and has an available range of `0.0` for silence, to `1.0` for unity gain.

Your application can set audio queue parameters in two ways:

- Per audio queue, using the `AudioQueueSetParameter` function. This lets you change settings for an audio queue directly. Such changes take effect immediately.
- Per audio queue buffer, using the `AudioQueueEnqueueBufferWithParameters` function. This lets you assign audio queue settings that are, in effect, carried by an audio queue buffer as you enqueue it. Such changes take effect when the audio queue buffer begins playing.

In both cases, parameter settings for an audio queue remain in effect until you change them.

You can access an audio queue's current parameter values at any time with the `AudioQueueGetParameter` function. See *Audio Queue Services Reference* for complete descriptions of the functions for getting and setting parameter values.

Recording Audio

When you record using Audio Queue Services, the destination can be just about anything—an on-disk file, a network connection, an object in memory, and so on. This chapter describes the most common scenario: basic recording to an on-disk file.

> **Note:** This chapter describes an ANSI-C based implementation for recording, with some use of C++ classes from the Mac OS X Core Audio SDK. For an Objective-C based example, see the *SpeakHere* sample code in the iOS Dev Center.

To add recording functionality to your application, you typically perform the following steps:

1. Define a custom structure to manage state, format, and path information.
2. Write an audio queue callback function to perform the actual recording.
3. Optionally write code to determine a good size for the audio queue buffers. Write code to work with magic cookies, if you'll be recording in a format that uses cookies.
4. Fill the fields of the custom structure. This includes specifying the data stream that the audio queue sends to the file it's recording into, as well as the path to that file.
5. Create a recording audio queue and ask it to create a set of audio queue buffers. Also create a file to record into.
6. Tell the audio queue to start recording.
7. When done, tell the audio queue to stop and then dispose of it. The audio queue disposes of its buffers.

The remainder of this chapter describes each of these steps in detail.

Define a Custom Structure to Manage State

The first step in developing a recording solution using Audio Queue Services is to define a custom structure. You'll use this structure to manage the audio format and audio queue state information. Listing 2-1 illustrates such a structure:

Listing 2-1 A custom structure for a recording audio queue

```
static const int kNumberBuffers = 3;                            // 1
struct AQRecorderState {
    AudioStreamBasicDescription  mDataFormat;                   // 2
    AudioQueueRef                mQueue;                        // 3
    AudioQueueBufferRef          mBuffers[kNumberBuffers];      // 4
    AudioFileID                  mAudioFile;                    // 5
    UInt32                       bufferByteSize;                // 6
    SInt64                       mCurrentPacket;                // 7
    bool                         mIsRunning;                    // 8
};
```

Here's a description of the fields in this structure:

1. Sets the number of audio queue buffers to use.
2. An `AudioStreamBasicDescription` structure (from `CoreAudioTypes.h`) representing the audio data format to write to disk. This format gets used by the audio queue specified in the `mQueue` field.

   The `mDataFormat` field gets filled initially by code in your program, as described in Set Up an Audio Format for Recording (page 32). It is good practice to then update the value of this field by querying the audio queue's `kAudioQueueProperty_StreamDescription` property, as described in Getting the Full Audio Format from an Audio Queue (page 35). On Mac OS X v10.5, use the `kAudioConverterCurrentInputStreamDescription` property instead.

   For details on the `AudioStreamBasicDescription` structure, see *Core Audio Data Types Reference*.
3. The recording audio queue created by your application.
4. An array holding pointers to the audio queue buffers managed by the audio queue.
5. An audio file object representing the file into which your program records audio data.
6. The size, in bytes, for each audio queue buffer. This value is calculated in these examples in the `DeriveBufferSize` function, after the audio queue is created and before it is started. See Write a Function to Derive Recording Audio Queue Buffer Size (page 29).
7. The packet index for the first packet to be written from the current audio queue buffer.
8. A Boolean value indicating whether or not the audio queue is running.

Write a Recording Audio Queue Callback

Next, write a recording audio queue callback function. This callback does two main things:

- Writes the contents of a newly filled audio queue buffer to the audio file you're recording into
- Enqueues the audio queue buffer (whose contents were just written to disk) to the buffer queue

This section shows an example callback declaration, then describes these two tasks separately, and finally presents an entire recording callback. For an illustration of the role of a recording audio queue callback, you can refer back to Figure 1-3 (page 13).

The Recording Audio Queue Callback Declaration

Listing 2-2 shows an example declaration for a recording audio queue callback function, declared as `AudioQueueInputCallback` in the `AudioQueue.h` header file:

Listing 2-2 The recording audio queue callback declaration

```
static void HandleInputBuffer (
    void                                *aqData,            // 1
    AudioQueueRef                       inAQ,               // 2
    AudioQueueBufferRef                 inBuffer,           // 3
    const AudioTimeStamp                *inStartTime,       // 4
    UInt32                              inNumPackets,       // 5
    const AudioStreamPacketDescription  *inPacketDesc       // 6
)
```

Here's how this code works:

1. Typically, `aqData` is a custom structure that contains state data for the audio queue, as described in Define a Custom Structure to Manage State (page 23).
2. The audio queue that owns this callback.
3. The audio queue buffer containing the incoming audio data to record.
4. The sample time of the first sample in the audio queue buffer (not needed for simple recording).
5. The number of packet descriptions in the `inPacketDesc` parameter. A value of `0` indicates CBR data.
6. For compressed audio data formats that require packet descriptions, the packet descriptions produced by the encoder for the packets in the buffer.

Writing an Audio Queue Buffer to Disk

The first task of a recording audio queue callback is to write an audio queue buffer to disk. This buffer is the one the callback's audio queue has just finished filling with new audio data from an input device. The callback uses the `AudioFileWritePackets` function from the `AudioFile.h` header file, as shown in Listing 2-3.

Listing 2-3 Writing an audio queue buffer to disk

```
AudioFileWritePackets (                     // 1
    pAqData->mAudioFile,                    // 2
    false,                                  // 3
    inBuffer->mAudioDataByteSize,           // 4
    inPacketDesc,                           // 5
    pAqData->mCurrentPacket,                // 6
    &inNumPackets,                          // 7
    inBuffer->mAudioData                    // 8
);
```

Here's how this code works:

1. The `AudioFileWritePackets` function, declared in the `AudioFile.h` header file, writes the contents of a buffer to an audio data file.
2. The audio file object (of type `AudioFileID`) that represents the audio file to write to. The `pAqData` variable is a pointer to the data structure described in Listing 2-1 (page 24).
3. Uses a value of `false` to indicate that the function should not cache the data when writing.
4. The number of bytes of audio data being written. The `inBuffer` variable represents the audio queue buffer handed to the callback by the audio queue.
5. An array of packet descriptions for the audio data. A value of `NULL` indicates no packet descriptions are required (such as for CBR audio data).
6. The packet index for the first packet to be written.
7. On input, the number of packets to write. On output, the number of packets actually written.
8. The new audio data to write to the audio file.

Enqueuing an Audio Queue Buffer

Now that the audio data from an audio queue buffer has been written to the audio file, the callback enqueues the buffer, as shown in Listing 2-4. Once back in the buffer queue, the buffer is in line and ready to accept more incoming audio data.

Listing 2-4 Enqueuing an audio queue buffer after writing to disk

```
AudioQueueEnqueueBuffer (                   // 1
    pAqData->mQueue,                        // 2
    inBuffer,                               // 3
    0,                                      // 4
    NULL                                    // 5
);
```

Here's how this code works:

1. The `AudioQueueEnqueueBuffer` function adds an audio queue buffer to an audio queue's buffer queue.
2. The audio queue to add the designated audio queue buffer to. The `pAqData` variable is a pointer to the data structure described in Listing 2-1.
3. The audio queue buffer to enqueue.
4. The number of packet descriptions in the audio queue buffer's data. Set to `0` because this parameter is unused for recording.
5. The array of packet descriptions describing the audio queue buffer's data. Set to `NULL` because this parameter is unused for recording.

A Full Recording Audio Queue Callback

Listing 2-5 shows a basic version of a full recording audio queue callback. As with the rest of the code examples in this document, this listing excludes error handling.

Listing 2-5 A recording audio queue callback function

```
static void HandleInputBuffer (
    void                                *aqData,
    AudioQueueRef                       inAQ,
    AudioQueueBufferRef                 inBuffer,
```

```
    const AudioTimeStamp                *inStartTime,
    UInt32                              inNumPackets,
    const AudioStreamPacketDescription  *inPacketDesc
) {
    AQRecorderState *pAqData = (AQRecorderState *) aqData;               // 1

    if (inNumPackets == 0 &&                                            // 2
          pAqData->mDataFormat.mBytesPerPacket != 0)
       inNumPackets =
           inBuffer->mAudioDataByteSize / pAqData->mDataFormat.mBytesPerPacket;

    if (AudioFileWritePackets (                                         // 3
            pAqData->mAudioFile,
            false,
            inBuffer->mAudioDataByteSize,
            inPacketDesc,
            pAqData->mCurrentPacket,
            &inNumPackets,
            inBuffer->mAudioData
        ) == noErr) {
            pAqData->mCurrentPacket += inNumPackets;                    // 4
    }
   if (pAqData->mIsRunning == 0)                                        // 5
      return;

    AudioQueueEnqueueBuffer (                                           // 6
        pAqData->mQueue,
        inBuffer,
        0,
        NULL
    );
}
```

Here's how this code works:

1. The custom structure supplied to the audio queue object upon instantiation, including an audio file object representing the audio file to record into as well as a variety of state data. See Define a Custom Structure to Manage State (page 23).
2. If the audio queue buffer contains CBR data, calculate the number of packets in the buffer. This number equals the total bytes of data in the buffer divided by the (constant) number of bytes per packet. For VBR data, the audio queue supplies the number of packets in the buffer when it invokes the callback.
3. Writes the contents of the buffer to the audio data file. For a detailed description , see Writing an Audio Queue Buffer to Disk (page 26).
4. If successful in writing the audio data, increment the audio data file's packet index to be ready for writing the next buffer's worth of audio data.
5. If the audio queue has stopped, return.
6. Enqueues the audio queue buffer whose contents have just been written to the audio file. For a detailed description, see Enqueuing an Audio Queue Buffer (page 27).

Write a Function to Derive Recording Audio Queue Buffer Size

Audio Queue Services expects your application to specify a size for the audio queue buffers you use. Listing 2-6 shows one way to do this. It derives a buffer size large enough to hold a given duration of audio data.

The calculation here takes into account the audio data format you're recording to. The format includes all the factors that might affect buffer size, such as the number of audio channels.

Listing 2-6 Deriving a recording audio queue buffer size

```
void DeriveBufferSize (
    AudioQueueRef                audioQueue,                 // 1
    AudioStreamBasicDescription  &ASBDescription,            // 2
    Float64                      seconds,                    // 3
    UInt32                       *outBufferSize              // 4
) {
    static const int maxBufferSize = 0x50000;                // 5

    int maxPacketSize = ASBDescription.mBytesPerPacket;      // 6
    if (maxPacketSize == 0) {                                // 7
        UInt32 maxVBRPacketSize = sizeof(maxPacketSize);
        AudioQueueGetProperty (
```

```
                audioQueue,
                kAudioQueueProperty_MaximumOutputPacketSize,
                // in Mac OS X v10.5, instead use
                //   kAudioConverterPropertyMaximumOutputPacketSize
                &maxPacketSize,
                &maxVBRPacketSize
        );
    }

    Float64 numBytesForTime =
        ASBDescription.mSampleRate * maxPacketSize * seconds; // 8
    *outBufferSize =
    UInt32 (numBytesForTime < maxBufferSize ?
        numBytesForTime : maxBufferSize);                    // 9
}
```

Here's how this code works:

1. The audio queue that owns the buffers whose size you want to specify.
2. The `AudioStreamBasicDescription` structure for the audio queue.
3. The size you are specifying for each audio queue buffer, in terms of seconds of audio.
4. On output, the size for each audio queue buffer, in terms of bytes.
5. An upper bound for the audio queue buffer size, in bytes. In this example, the upper bound is set to 320 KB. This corresponds to approximately five seconds of stereo, 24 bit audio at a sample rate of 96 kHz.
6. For CBR audio data, get the (constant) packet size from the `AudioStreamBasicDescription` structure. Use this value as the maximum packet size.

   This assignment has the side effect of determining if the audio data to be recorded is CBR or VBR. If it is VBR, the audio queue's `AudioStreamBasicDescription` structure lists the value of bytes-per-packet as `0`.
7. For VBR audio data, query the audio queue to get the estimated maximum packet size.
8. Derive the buffer size, in bytes.
9. Limit the buffer size, if needed, to the previously set upper bound.

Set a Magic Cookie for an Audio File

Some compressed audio formats, such as MPEG 4 AAC, make use of structures that contain audio metadata. These structures are called **magic cookies**. When you record to such a format using Audio Queue Services, you must get the magic cookie from the audio queue and add it to the audio file before you start recording.

Listing 2-7 shows how to obtain a magic cookie from an audio queue and apply it to an audio file. Your code would call a function like this before recording, and then again after recording—some codecs update magic cookie data when recording has stopped.

Listing 2-7 Setting a magic cookie for an audio file

```
OSStatus SetMagicCookieForFile (
    AudioQueueRef inQueue,                                  // 1
    AudioFileID   inFile                                    // 2
) {
    OSStatus result = noErr;                                // 3
    UInt32 cookieSize;                                      // 4

    if (
            AudioQueueGetPropertySize (                     // 5
                inQueue,
                kAudioQueueProperty_MagicCookie,
                &cookieSize
            ) == noErr
    ) {
        char* magicCookie =
            (char *) malloc (cookieSize);                   // 6
        if (
                AudioQueueGetProperty (                     // 7
                    inQueue,
                    kAudioQueueProperty_MagicCookie,
                    magicCookie,
                    &cookieSize
                ) == noErr
        )
            result =    AudioFileSetProperty (              // 8
                            inFile,
```

```
                            kAudioFilePropertyMagicCookieData,
                            cookieSize,
                            magicCookie
                        );
        free (magicCookie);                                     // 9
    }
    return result;                                              // 10
}
```

Here's how this code works:

1. The audio queue you're using for recording.
2. The audio file you're recording into.
3. A result variable that indicates the success or failure of this function.
4. A variable to hold the magic cookie data size.
5. Gets the data size of the magic cookie from the audio queue and stores it in the `cookieSize` variable.
6. Allocates an array of bytes to hold the magic cookie information.
7. Gets the magic cookie by querying the audio queue's `kAudioQueueProperty_MagicCookie` property.
8. Sets the magic cookie for the audio file you're recording into. The `AudioFileSetProperty` function is declared in the `AudioFile.h` header file.
9. Frees the memory for the temporary cookie variable.
10. Returns the success or failure of this function.

Set Up an Audio Format for Recording

This section describes how you set up an audio data format for the audio queue. The audio queue uses this format for recording to a file.

To set up an audio data format, you specify:

- Audio data format type (such as linear PCM, AAC, etc.)
- Sample rate (such as 44.1 kHz)
- Number of audio channels (such as 2, for stereo)
- Bit depth (such as 16 bits)

- Frames per packet (linear PCM, for example, uses one frame per packet)
- Audio file type (such as CAF, AIFF, etc.)
- Details of the audio data format required for the file type

Listing 2-8 illustrates setting up an audio format for recording, using a fixed choice for each attribute. In production code, you'd typically allow the user to specify some or all aspects of the audio format. With either approach, the goal is to fill the `mDataFormat` field of the `AQRecorderState` custom structure, described in Define a Custom Structure to Manage State (page 23).

Listing 2-8 Specifying an audio queue's audio data format

```
AQRecorderState aqData;                                  // 1

aqData.mDataFormat.mFormatID         = kAudioFormatLinearPCM; // 2
aqData.mDataFormat.mSampleRate       = 44100.0;              // 3
aqData.mDataFormat.mChannelsPerFrame = 2;                    // 4
aqData.mDataFormat.mBitsPerChannel   = 16;                   // 5
aqData.mDataFormat.mBytesPerPacket   =                       // 6
   aqData.mDataFormat.mBytesPerFrame =
      aqData.mDataFormat.mChannelsPerFrame * sizeof (SInt16);
aqData.mDataFormat.mFramesPerPacket  = 1;                    // 7

AudioFileTypeID fileType             = kAudioFileAIFFType;    // 8
aqData.mDataFormat.mFormatFlags =                            // 9
    kLinearPCMFormatFlagIsBigEndian
    | kLinearPCMFormatFlagIsSignedInteger
    | kLinearPCMFormatFlagIsPacked;
```

Here's how this code works:

1. Creates an instance of the `AQRecorderState` custom structure. The structure's `mDataFormat` field contains an `AudioStreamBasicDescription` structure. The values set in the `mDataFormat` field provide an initial definition of the audio format for the audio queue—which is also the audio format for the file you record into. In Listing 2-10 (page 35), you obtain a more complete specification of the audio format, which Core Audio provides to you based on the format type and file type.
2. Defines the audio data format type as linear PCM. See *Core Audio Data Types Reference* for a complete listing of the available data formats.

3. Defines the sample rate as 44.1 kHz.
4. Defines the number of channels as 2.
5. Defines the bit depth per channel as 16.
6. Defines the number of bytes per packet, and the number of bytes per frame, to 4 (that is, 2 channels times 2 bytes per sample).
7. Defines the number of frames per packet as 1.
8. Defines the file type as AIFF. See the audio file types enumeration in the `AudioFile.h` header file for a complete listing of the available file types. You can specify any file type for which there is an installed codec, as described in Using Codecs and Audio Data Formats (page 18).
9. Sets the format flags needed for the specified file type.

Create a Recording Audio Queue

Now, with the recording callback and audio data format set up, you create and configure an audio queue for recording.

Creating a Recording Audio Queue

Listing 2-9 illustrates how to create a recording audio queue. Notice that the `AudioQueueNewInput` function uses the callback, the custom structure, and the audio data format that were configured in previous steps.

Listing 2-9 Creating a recording audio queue

```
AudioQueueNewInput (                            // 1
    &aqData.mDataFormat,                        // 2
    HandleInputBuffer,                          // 3
    &aqData,                                    // 4
    NULL,                                       // 5
    kCFRunLoopCommonModes,                      // 6
    0,                                          // 7
    &aqData.mQueue                              // 8
);
```

Here's how this code works:

1. The `AudioQueueNewInput` function creates a new recording audio queue.

2. The audio data format to use for the recording. See Set Up an Audio Format for Recording (page 32).
3. The callback function to use with the recording audio queue. See Write a Recording Audio Queue Callback (page 25).
4. The custom data structure for the recording audio queue. See Define a Custom Structure to Manage State (page 23).
5. The run loop on which the callback will be invoked. Use `NULL` to specify default behavior, in which the callback will be invoked on a thread internal to the audio queue. This is typical use—it allows the audio queue to record while your application's user interface thread waits for user input to stop the recording.
6. The run loop modes in which the callback can be invoked. Normally, use the `kCFRunLoopCommonModes` constant here.
7. Reserved. Must be `0`.
8. On output, the newly allocated recording audio queue.

Getting the Full Audio Format from an Audio Queue

When the audio queue came into existence (see Creating a Recording Audio Queue (page 34)), it may have filled out the `AudioStreamBasicDescription` structure more completely than you have, particularly for compressed formats. To obtain the complete format description, call the `AudioQueueGetProperty` function as shown in Listing 2-10. You use the complete audio format when you create an audio file to record into (see Create an Audio File (page 36)).

Listing 2-10 Getting the audio format from an audio queue

```
UInt32 dataFormatSize = sizeof (aqData.mDataFormat);       // 1

AudioQueueGetProperty (                                    // 2
    aqData.mQueue,                                         // 3
    kAudioQueueProperty_StreamDescription,                 // 4
    // in Mac OS X, instead use
    //    kAudioConverterCurrentInputStreamDescription
    &aqData.mDataFormat,                                   // 5
    &dataFormatSize                                        // 6
);
```

Here's how this code works:

1. Gets an expected property value size to use when querying the audio queue about its audio data format.

2. The `AudioQueueGetProperty` function obtains the value for a specified property in an audio queue.
3. The audio queue to obtain the audio data format from.
4. The property ID for obtaining the value of the audio queue's data format.
5. On output, the full audio data format, in the form of an `AudioStreamBasicDescription` structure, obtained from the audio queue.
6. On input, the expected size of the `AudioStreamBasicDescription` structure. On output, the actual size. Your recording application does not need to make use of this value.

Create an Audio File

With an audio queue created and configured, you create the audio file that you'll record audio data into, as shown in Listing 2-11. The audio file uses the data format and file format specifications previously stored in the audio queue's custom structure.

Listing 2-11 Creating an audio file for recording

```
CFURLRef audioFileURL =
    CFURLCreateFromFileSystemRepresentation (            // 1
        NULL,                                            // 2
        (const UInt8 *) filePath,                        // 3
        strlen (filePath),                               // 4
        false                                            // 5
    );

AudioFileCreateWithURL (                                 // 6
    audioFileURL,                                        // 7
    fileType,                                            // 8
    &aqData.mDataFormat,                                 // 9
    kAudioFileFlags_EraseFile,                           // 10
    &aqData.mAudioFile                                   // 11
);
```

Here's how this code works:

1. The `CFURLCreateFromFileSystemRepresentation` function, declared in the `CFURL.h` header file, creates a CFURL object representing a file to record into.

2. Use `NULL` (or `kCFAllocatorDefault`) to use the current default memory allocator.
3. The file-system path you want to convert to a CFURL object. In production code, you would typically obtain a value for `filePath` from the user.
4. The number of bytes in the file-system path.
5. A value of `false` indicates that `filePath` represents a file, not a directory.
6. The `AudioFileCreateWithURL` function, from the `AudioFile.h` header file, creates a new audio file or initializes an existing file.
7. The URL at which to create the new audio file, or to initialize in the case of an existing file. The URL was derived from the `CFURLCreateFromFileSystemRepresentation` in step 1.
8. The file type for the new file. In the example code in this chapter, this was previously set to AIFF by way of the `kAudioFileAIFFType` file type constant. See Set Up an Audio Format for Recording (page 32).
9. The data format of the audio that will be recorded into the file, specified as an `AudioStreamBasicDescription` structure. In the example code for this chapter, this was also set in Set Up an Audio Format for Recording (page 32).
10. Erases the file, in the case that the file already exists.
11. On output, an audio file object (of type `AudioFileID`) representing the audio file to record into.

Set an Audio Queue Buffer Size

Before you prepare a set of audio queue buffers that you'll use while recording, you make use of the `DeriveBufferSize` function you wrote earlier (see Write a Function to Derive Recording Audio Queue Buffer Size (page 29)). You assign this size to the recording audio queue you are using. Listing 2-12 illustrates this:

Listing 2-12 Setting an audio queue buffer size

```
DeriveBufferSize (                              // 1
    aqData.mQueue,                              // 2
    aqData.mDataFormat,                         // 3
    0.5,                                        // 4
    &aqData.bufferByteSize                      // 5
);
```

Here's how this code works:

1. The `DeriveBufferSize` function, described in Write a Function to Derive Recording Audio Queue Buffer Size (page 29), sets an appropriate audio queue buffer size.

2. The audio queue that you're setting buffer size for.
3. The audio data format for the file you are recording. See Set Up an Audio Format for Recording (page 32).
4. The number of seconds of audio that each audio queue buffer should hold. One half second, as set here, is typically a good choice.
5. On output, the size for each audio queue buffer, in bytes. This value is placed in the custom structure for the audio queue.

Prepare a Set of Audio Queue Buffers

You now ask the audio queue that you've created (in Create a Recording Audio Queue (page 34)) to prepare a set of audio queue buffers. Listing 2-13 demonstrates how to do this.

Listing 2-13 Preparing a set of audio queue buffers

```
for (int i = 0; i < kNumberBuffers; ++i) {           // 1
    AudioQueueAllocateBuffer (                       // 2
        aqData.mQueue,                               // 3
        aqData.bufferByteSize,                       // 4
        &aqData.mBuffers[i]                          // 5
    );

    AudioQueueEnqueueBuffer (                        // 6
        aqData.mQueue,                               // 7
        aqData.mBuffers[i],                          // 8
        0,                                           // 9
        NULL                                         // 10
    );
}
```

Here's how this code works:

1. Iterates to allocate and enqueue each audio queue buffer.
2. The `AudioQueueAllocateBuffer` function asks an audio queue to allocate an audio queue buffer.
3. The audio queue that performs the allocation and that will own the buffer.

4. The size, in bytes, for the new audio queue buffer being allocated. See Write a Function to Derive Recording Audio Queue Buffer Size (page 29).
5. On output, the newly allocated audio queue buffer. The pointer to the buffer is placed in the custom structure you're using with the audio queue.
6. The `AudioQueueEnqueueBuffer` function adds an audio queue buffer to the end of a buffer queue.
7. The audio queue whose buffer queue you are adding the buffer to.
8. The audio queue buffer you are enqueuing.
9. This parameter is unused when enqueuing a buffer for recording.
10. This parameter is unused when enqueuing a buffer for recording.

Record Audio

All of the preceding code has led up to the very simple process of recording, as shown in Listing 2-14.

Listing 2-14 Recording audio

```
aqData.mCurrentPacket = 0;                                  // 1
aqData.mIsRunning = true;                                   // 2

AudioQueueStart (                                           // 3
    aqData.mQueue,                                          // 4
    NULL                                                    // 5
);
// Wait, on user interface thread, until user stops the recording
AudioQueueStop (                                            // 6
    aqData.mQueue,                                          // 7
    true                                                    // 8
);

aqData.mIsRunning = false;                                  // 9
```

Here's how this code works:

1. Initializes the packet index to `0` to begin recording at the start of the audio file.
2. Sets a flag in the custom structure to indicate that the audio queue is running. This flag is used by the recording audio queue callback.

3. The `AudioQueueStart` function starts the audio queue, on its own thread.
4. The audio queue to start.
5. Uses `NULL` to indicate that the audio queue should start recording immediately.
6. The `AudioQueueStop` function stops and resets the recording audio queue.
7. The audio queue to stop.
8. Use `true` to use synchronous stopping. See Audio Queue Control and State (page 20) for an explanation of synchronous and asynchronous stopping.
9. Sets a flag in the custom structure to indicate that the audio queue is not running.

Clean Up After Recording

When you're finished with recording, dispose of the audio queue and close the audio file. Listing 2-15 illustrates these steps.

Listing 2-15 Cleaning up after recording

```
AudioQueueDispose (                                 // 1
    aqData.mQueue,                                  // 2
    true                                            // 3
);

AudioFileClose (aqData.mAudioFile);                 // 4
```

Here's how this code works:

1. The `AudioQueueDispose` function disposes of the audio queue and all of its resources, including its buffers.
2. The audio queue you want to dispose of.
3. Use `true` to dispose of the audio queue synchronously (that is, immediately).
4. Closes the audio file that was used for recording. The `AudioFileClose` function is declared in the `AudioFile.h` header file.

Playing Audio

When you play audio using Audio Queue Services, the source can be just about anything—an on-disk file, a software-based audio synthesizer, an object in memory, and so on. This chapter describes the most common scenario: playing back an on-disk file.

> **Note:** This chapter describes an ANSI-C based implementation for playback, with some use of C++ classes from the Mac OS X Core Audio SDK. For an Objective-C based example, see the *SpeakHere* sample code in the iOS Dev Center.

To add playback functionality to your application, you typically perform the following steps:

1. Define a custom structure to manage state, format, and path information.
2. Write an audio queue callback function to perform the actual playback.
3. Write code to determine a good size for the audio queue buffers.
4. Open an audio file for playback and determine its audio data format.
5. Create a playback audio queue and configure it for playback.
6. Allocate and enqueue audio queue buffers. Tell the audio queue to start playing. When done, the playback callback tells the audio queue to stop.
7. Dispose of the audio queue. Release resources.

The remainder of this chapter describes each of these steps in detail.

Define a Custom Structure to Manage State

To start, define a custom structure that you'll use to manage audio format and audio queue state information. Listing 3-1 illustrates such a structure:

Listing 3-1 A custom structure for a playback audio queue

```
static const int kNumberBuffers = 3;                              // 1
struct AQPlayerState {
```

```
    AudioStreamBasicDescription   mDataFormat;                    // 2
    AudioQueueRef                 mQueue;                         // 3
    AudioQueueBufferRef           mBuffers[kNumberBuffers];       // 4
    AudioFileID                   mAudioFile;                     // 5
    UInt32                        bufferByteSize;                 // 6
    SInt64                        mCurrentPacket;                 // 7
    UInt32                        mNumPacketsToRead;              // 8
    AudioStreamPacketDescription  *mPacketDescs;                  // 9
    bool                          mIsRunning;                     // 10
};
```

Most fields in this structure are identical (or nearly so) to those in the custom structure used for recording, as described in the Recording Audio chapter in Define a Custom Structure to Manage State (page 23). For example, the `mDataFormat` field is used here to hold the format of the file being played. When recording, the analogous field holds the format of the file being written to disk.

Here's a description of the fields in this structure:

1. Sets the number of audio queue buffers to use. Three is typically a good number, as described in Audio Queue Buffers (page 11).
2. An `AudioStreamBasicDescription` structure (from `CoreAudioTypes.h`) representing the audio data format of the file being played. This format gets used by the audio queue specified in the `mQueue` field.

   The `mDataFormat` field gets filled by querying an audio file's `kAudioFilePropertyDataFormat` property, as described in Obtaining a File's Audio Data Format (page 51).

   For details on the `AudioStreamBasicDescription` structure, see *Core Audio Data Types Reference*.
3. The playback audio queue created by your application.
4. An array holding pointers to the audio queue buffers managed by the audio queue.
5. An audio file object that represents the audio file your program plays.
6. The size, in bytes, for each audio queue buffer. This value is calculated in these examples in the `DeriveBufferSize` function, after the audio queue is created and before it is started. See Write a Function to Derive Playback Audio Queue Buffer Size (page 47).
7. The packet index for the next packet to play from the audio file.
8. The number of packets to read on each invocation of the audio queue's playback callback. Like the `bufferByteSize` field, this value is calculated in these examples in the `DeriveBufferSize` function, after the audio queue is created and before it is started.

9. For VBR audio data, the array of packet descriptions for the file being played. For CBR data, the value of this field is `NULL`.
10. A Boolean value indicating whether or not the audio queue is running.

Write a Playback Audio Queue Callback

Next, write a playback audio queue callback function. This callback does three main things:

- Reads a specified amount of data from an audio file and puts it into an audio queue buffer
- Enqueues the audio queue buffer to the buffer queue
- When there's no more data to read from the audio file, tells the audio queue to stop

This section shows an example callback declaration, describes each of these tasks separately, and finally presents an entire playback callback. For an illustration of the role of a playback callback, you can refer back to Figure 1-4 (page 15).

The Playback Audio Queue Callback Declaration

Listing 3-2 shows an example declaration for a playback audio queue callback function, declared as `AudioQueueOutputCallback` in the `AudioQueue.h` header file:

Listing 3-2 The playback audio queue callback declaration

```
static void HandleOutputBuffer (
    void                *aqData,                 // 1
    AudioQueueRef       inAQ,                    // 2
    AudioQueueBufferRef  inBuffer                // 3
)
```

Here's how this code works:

1. Typically, `aqData` is the custom structure that contains state information for the audio queue, as described in Define a Custom Structure to Manage State (page 41).
2. The audio queue that owns this callback.
3. An audio queue buffer that the callback is to fill with data by reading from an audio file.

Reading From a File into an Audio Queue Buffer

The first action of a playback audio queue callback is to read data from an audio file and place it in an audio queue buffer. Listing 3-3 shows how to do this.

Listing 3-3 Reading from an audio file into an audio queue buffer

```
AudioFileReadPackets (                          // 1
    pAqData->mAudioFile,                        // 2
    false,                                      // 3
    &numBytesReadFromFile,                      // 4
    pAqData->mPacketDescs,                      // 5
    pAqData->mCurrentPacket,                    // 6
    &numPackets,                                // 7
    inBuffer->mAudioData                        // 8
);
```

Here's how this code works:

1. The `AudioFileReadPackets` function, declared in the `AudioFile.h` header file, reads data from an audio file and places it into a buffer.
2. The audio file to read from.
3. Uses a value of `false` to indicate that the function should not cache the data when reading.
4. On output, the number of bytes of audio data that were read from the audio file.
5. On output, an array of packet descriptions for the data that was read from the audio file. For CBR data, the input value of this parameter is `NULL`.
6. The packet index for the first packet to read from the audio file.
7. On input, the number of packets to read from the audio file. On output, the number of packets actually read.
8. On output, the filled audio queue buffer containing data that was read from the audio file.

Enqueuing an Audio Queue Buffer

Now that data has been read from an audio file and placed in an audio queue buffer, the callback enqueues the buffer, as shown in Listing 3-4. Once in the buffer queue, the audio data in the buffer is available for the audio queue to send to the output device.

Listing 3-4 Enqueuing an audio queue buffer after reading from disk

```
AudioQueueEnqueueBuffer (                       // 1
    pAqData->mQueue,                            // 2
    inBuffer,                                   // 3
    (pAqData->mPacketDescs ? numPackets : 0),   // 4
    pAqData->mPacketDescs                       // 5
);
```

Here's how this code works:

1. The `AudioQueueEnqueueBuffer` function adds an audio queue buffer to a buffer queue.
2. The audio queue that owns the buffer queue.
3. The audio queue buffer to enqueue
4. The number of packets represented in the audio queue buffer's data. For CBR data, which uses no packet descriptions, uses `0`.
5. For compressed audio data formats that use packet descriptions, the packet descriptions for the packets in the buffer. .

Stopping an Audio Queue

The last thing your callback does is to check if there's no more data to read from the audio file that you're playing. Upon discovering the end of the file, your callback tells the playback audio queue to stop. Listing 3-5 illustrates this.

Listing 3-5 Stopping an audio queue

```
if (numPackets == 0) {                          // 1
    AudioQueueStop (                            // 2
        pAqData->mQueue,                        // 3
        false                                   // 4
    );
    pAqData->mIsRunning = false;                // 5
}
```

Here's how this code works:

1. Checks if the number of packets read by the `AudioFileReadPackets` function (invoked earlier by the callback) is `0`.
2. The `AudioQueueStop` function stops the audio queue.
3. The audio queue to stop.
4. Stops the audio queue asynchronously, when all queued buffers have been played. See Audio Queue Control and State (page 20).
5. Sets a flag in the custom structure to indicate that playback is finished.

A Full Playback Audio Queue Callback

Listing 3-6 shows a basic version of a full playback audio queue callback. As with the rest of the code examples in this document, this listing excludes error handling.

Listing 3-6 A playback audio queue callback function

```
static void HandleOutputBuffer (
    void                *aqData,
    AudioQueueRef       inAQ,
    AudioQueueBufferRef inBuffer
) {
    AQPlayerState *pAqData = (AQPlayerState *) aqData;        // 1
    if (pAqData->mIsRunning == 0) return;                    // 2
    UInt32 numBytesReadFromFile;                             // 3
    UInt32 numPackets = pAqData->mNumPacketsToRead;          // 4
    AudioFileReadPackets (
        pAqData->mAudioFile,
        false,
        &numBytesReadFromFile,
        pAqData->mPacketDescs,
        pAqData->mCurrentPacket,
        &numPackets,
        inBuffer->mAudioData
    );
    if (numPackets > 0) {                                    // 5
        inBuffer->mAudioDataByteSize = numBytesReadFromFile; // 6
       AudioQueueEnqueueBuffer (
            pAqData->mQueue,
            inBuffer,
            (pAqData->mPacketDescs ? numPackets : 0),
            pAqData->mPacketDescs
        );
        pAqData->mCurrentPacket += numPackets;               // 7
    } else {
        AudioQueueStop (
            pAqData->mQueue,
            false
```

```
        );
        pAqData->mIsRunning = false;
    }
}
```

Here's how this code works:

1. The custom data supplied to the audio queue upon instantiation, including the audio file object (of type `AudioFileID`) representing the file to play as well as a variety of state data. See Define a Custom Structure to Manage State (page 41).
2. If the audio queue is stopped, returns immediately.
3. A variable to hold the number of bytes of audio data read from the file being played.
4. Initializes the `numPackets` variable with the number of packets to read from the file being played.
5. Tests whether some audio data was retrieved from the file. If so, enqueues the newly-filled buffer. If not, stops the audio queue.
6. Tells the audio queue buffer structure the number of bytes of data that were read.
7. Increments the packet index according to the number of packets that were read.

Write a Function to Derive Playback Audio Queue Buffer Size

Audio Queue Services expects your application to specify a size for the audio queue buffers you use. Listing 3-7 shows one way to do this. It derives a buffer size large enough to hold a given duration of audio data.

You'll call this `DeriveBufferSize` function in your application, after creating a playback audio queue, as a prerequisite to asking the audio queue to allocate buffers. See Set Sizes for a Playback Audio Queue (page 53).

The code here does two additional things compared to the analogous function you saw in Write a Function to Derive Recording Audio Queue Buffer Size (page 29). For playback you also:

- Derive the number of packets to read each time your callback invokes the `AudioFileReadPackets` function
- Set a lower bound on buffer size, to avoid overly frequent disk access

The calculation here takes into account the audio data format you're reading from disk. The format includes all the factors that might affect buffer size, such as the number of audio channels.

Listing 3-7 Deriving a playback audio queue buffer size

```
void DeriveBufferSize (
    AudioStreamBasicDescription &ASBDesc,                           // 1
    UInt32                      maxPacketSize,                      // 2
    Float64                     seconds,                            // 3
    UInt32                      *outBufferSize,                     // 4
    UInt32                      *outNumPacketsToRead                // 5
) {
    static const int maxBufferSize = 0x50000;                       // 6
    static const int minBufferSize = 0x4000;                        // 7

    if (ASBDesc.mFramesPerPacket != 0) {                            // 8
        Float64 numPacketsForTime =
            ASBDesc.mSampleRate / ASBDesc.mFramesPerPacket * seconds;
        *outBufferSize = numPacketsForTime * maxPacketSize;
    } else {                                                        // 9
        *outBufferSize =
            maxBufferSize > maxPacketSize ?
                maxBufferSize : maxPacketSize;
    }

    if (                                                            // 10
        *outBufferSize > maxBufferSize &&
        *outBufferSize > maxPacketSize
    )
        *outBufferSize = maxBufferSize;
    else {                                                          // 11
        if (*outBufferSize < minBufferSize)
            *outBufferSize = minBufferSize;
    }

    *outNumPacketsToRead = *outBufferSize / maxPacketSize;          // 12
}
```

Here's how this code works:

1. The `AudioStreamBasicDescription` structure for the audio queue.
2. The estimated maximum packet size for the data in the audio file you're playing. You can determine this value by invoking the `AudioFileGetProperty` function (declared in the `AudioFile.h` header file) with a property ID of `kAudioFilePropertyPacketSizeUpperBound`. See Set Sizes for a Playback Audio Queue (page 53).
3. The size you are specifying for each audio queue buffer, in terms of seconds of audio.
4. On output, the size for each audio queue buffer, in bytes.
5. On output, the number of packets of audio data to read from the file on each invocation of the playback audio queue callback.
6. An upper bound for the audio queue buffer size, in bytes. In this example, the upper bound is set to 320 KB. This corresponds to approximately five seconds of stereo, 24 bit audio at a sample rate of 96 kHz.
7. A lower bound for the audio queue buffer size, in bytes. In this example, the lower bound is set to 16 KB.
8. For audio data formats that define a fixed number of frames per packet, derives the audio queue buffer size.
9. For audio data formats that do not define a fixed number of frames per packet, derives a reasonable audio queue buffer size based on the maximum packet size and the upper bound you've set.
10. If the derived buffer size is above the upper bound you've set, adjusts it the bound—taking into account the estimated maximum packet size.
11. If the derived buffer size is below the lower bound you've set, adjusts it to the bound.
12. Calculates the number of packets to read from the audio file on each invocation of the callback.

Open an Audio File for Playback

Now you open an audio file for playback, using these three steps:

1. Obtain a CFURL object representing the audio file you want to play.
2. Open the file.
3. Obtain the file's audio data format.

Obtaining a CFURL Object for an Audio File

Listing 3-8 demonstrates how to obtain a CFURL object for the audio file you want to play. You use the CFURL object in the next step, opening the file..

Listing 3-8 Obtaining a CFURL object for an audio file

```
CFURLRef audioFileURL =
    CFURLCreateFromFileSystemRepresentation (           // 1
        NULL,                                           // 2
        (const UInt8 *) filePath,                       // 3
        strlen (filePath),                              // 4
        false                                           // 5
    );
```

Here's how this code works:

1. The `CFURLCreateFromFileSystemRepresentation` function, declared in the `CFURL.h` header file, creates a CFURL object representing the file to play.
2. Uses `NULL` (or `kCFAllocatorDefault`) to use the current default memory allocator.
3. The file-system path you want to convert to a CFURL object. In production code, you would typically obtain a value for `filePath` from the user.
4. The number of bytes in the file-system path.
5. A value of `false` indicates that `filePath` represents a file, not a directory.

Opening an Audio File

Listing 3-9 demonstrates how to open an audio file for playback.

Listing 3-9 Opening an audio file for playback

```
AQPlayerState aqData;                                   // 1

OSStatus result =
    AudioFileOpenURL (                                  // 2
        audioFileURL,                                   // 3
        fsRdPerm,                                       // 4
        0,                                              // 5
        &aqData.mAudioFile                              // 6
    );

CFRelease (audioFileURL);                               // 7
```

Here's how this code works:

1. Creates an instance of the `AQPlayerState` custom structure (see Define a Custom Structure to Manage State (page 41)). You use this instance when you open an audio file for playback, as a place to hold the audio file object (of type `AudioFileID`) that represents the audio file.
2. The `AudioFileOpenURL` function, declared in the `AudioFile.h` header file, opens the file you want to play.
3. A reference to the file to play.
4. The file permissions you want to use with the file you're playing. The available permissions are defined in the File Manager's `File Access Permission Constants` enumeration. In this example you request permission to read the file.
5. An optional file type hint. A value of `0` here indicates that the example does not use this facility.
6. On output, a reference to the audio file is placed in the custom structure's `mAudioFile` field.
7. Releases the CFURL object that was created in step 1.

Obtaining a File's Audio Data Format

Listing 3-10 shows how to obtain a file's audio data format.

Listing 3-10 Obtaining a file's audio data format

```
UInt32 dataFormatSize = sizeof (aqData.mDataFormat);    // 1

AudioFileGetProperty (                                  // 2
    aqData.mAudioFile,                                  // 3
    kAudioFilePropertyDataFormat,                       // 4
    &dataFormatSize,                                    // 5
    &aqData.mDataFormat                                 // 6
);
```

Here's how this code works:

1. Gets an expected property value size to use when querying the audio file about its audio data format.
2. The `AudioFileGetProperty` function, declared in the `AudioFile.h` header file, obtains the value for a specified property in an audio file.

3. An audio file object (of type `AudioFileID`) representing the file whose audio data format you want to obtain.
4. The property ID for obtaining the value of the audio file's data format.
5. On input, the expected size of the `AudioStreamBasicDescription` structure that describes the audio file's data format. On output, the actual size. Your playback application does not need to make use of this value.
6. On output, the full audio data format, in the form of an `AudioStreamBasicDescription` structure, obtained from the audio file. This line applies the file's audio data format to the audio queue by storing it in the audio queue's custom structure.

Create a Playback Audio Queue

Listing 3-11 shows how to create a playback audio queue. Notice that the `AudioQueueNewOutput` function uses the custom structure and the callback that were configured in previous steps, as well as the audio data format of the file to be played.

Listing 3-11 Creating a playback audio queue

```
AudioQueueNewOutput (                                   // 1
    &aqData.mDataFormat,                                // 2
    HandleOutputBuffer,                                 // 3
    &aqData,                                            // 4
    CFRunLoopGetCurrent (),                             // 5
    kCFRunLoopCommonModes,                              // 6
    0,                                                  // 7
    &aqData.mQueue                                      // 8
);
```

Here's how this code works:

1. The `AudioQueueNewOutput` function creates a new playback audio queue.
2. The audio data format of the file that the audio queue is being set up to play. See Obtaining a File's Audio Data Format (page 51).
3. The callback function to use with the playback audio queue. See Write a Playback Audio Queue Callback (page 43).
4. The custom data structure for the playback audio queue. See Define a Custom Structure to Manage State (page 41).
5. The current run loop, and the one on which the audio queue playback callback will be invoked.

6. The run loop modes in which the callback can be invoked. Normally, use the `kCFRunLoopCommonModes` constant here.
7. Reserved. Must be `0`.
8. On output, the newly allocated playback audio queue.

Set Sizes for a Playback Audio Queue

Next, you set some sizes for the playback audio queue. You use these sizes when you allocate buffers for an audio queue and before you start reading an audio file.

The code listings in this section show how to set:

- Audio queue buffer size
- Number of packets to read for each invocation of the playback audio queue callback
- Array size for holding the packet descriptions for one buffer's worth of audio data

Setting Buffer Size and Number of Packets to Read

Listing 3-12 demonstrates how to use the `DeriveBufferSize` function you wrote earlier (see Write a Function to Derive Playback Audio Queue Buffer Size (page 47)). The goal here is to set a size, in bytes, for each audio queue buffer, and to determine the number of packets to read for each invocation of the playback audio queue callback.

This code uses a conservative estimate of maximum packet size, which Core Audio provides by way of the `kAudioFilePropertyPacketSizeUpperBound` property. In most cases, it is better to use this technique—which is approximate but fast—than to take the time to read an entire audio file to obtain the actual maximum packet size.

Listing 3-12 Setting playback audio queue buffer size and number of packets to read

```
UInt32 maxPacketSize;
UInt32 propertySize = sizeof (maxPacketSize);
AudioFileGetProperty (                                  // 1
    aqData.mAudioFile,                                  // 2
    kAudioFilePropertyPacketSizeUpperBound,             // 3
    &propertySize,                                      // 4
    &maxPacketSize                                      // 5
);
```

```
DeriveBufferSize (                              // 6
    aqData.mDataFormat,                         // 7
    maxPacketSize,                              // 8
    0.5,                                        // 9
    &aqData.bufferByteSize,                     // 10
    &aqData.mNumPacketsToRead                   // 11
);
```

Here's how this code works:

1. The `AudioFileGetProperty` function, declared in the `AudioFile.h` header file, obtains the value of a specified property for an audio file. Here you use it to get a conservative upper bound, in bytes, for the size of the audio data packets in the file you want to play.
2. An audio file object (of type `AudioFileID`) representing the file you want to play. See Opening an Audio File (page 50).
3. The property ID for obtaining a conservative upper bound for packet size in an audio file.
4. On output, the size, in bytes, for the `kAudioFilePropertyPacketSizeUpperBound` property.
5. On output, a conservative upper bound for packet size, in bytes, for the file you want to play.
6. The `DeriveBufferSize` function, described in Write a Function to Derive Playback Audio Queue Buffer Size (page 47), sets a buffer size and a number of packets to read on each invocation of the playback audio queue callback.
7. The audio data format of the file you want to play. See Obtaining a File's Audio Data Format (page 51).
8. The estimated maximum packet size in the audio file, from line 5 of this listing.
9. The number of seconds of audio that each audio queue buffer should hold. One half second, as set here, is typically a good choice.
10. On output, the size for each audio queue buffer, in bytes. This value is placed in the custom structure for the audio queue.
11. On output, the number of packets to read on each invocation of the playback audio queue callback. This value is also placed in the custom structure for the audio queue.

Allocating Memory for a Packet Descriptions Array

Now you allocate memory for an array to hold the packet descriptions for one buffer's worth of audio data. Constant bitrate data does not use packet descriptions, so the CBR case—step 3 in Listing 3-13—is very simple.

Listing 3-13 Allocating memory for a packet descriptions array

```
bool isFormatVBR = (                                        // 1
    aqData.mDataFormat.mBytesPerPacket == 0 ||
    aqData.mDataFormat.mFramesPerPacket == 0
);

if (isFormatVBR) {                                          // 2
    aqData.mPacketDescs =
      (AudioStreamPacketDescription*) malloc (
        aqData.mNumPacketsToRead * sizeof (AudioStreamPacketDescription)
      );
} else {                                                    // 3
    aqData.mPacketDescs = NULL;
}
```

Here's how this code works:

1. Determines if the audio file's data format is VBR or CBR. In VBR data, one or both of the bytes-per-packet or frames-per-packet values is variable, and so will be listed as `0` in the audio queue's `AudioStreamBasicDescription` structure.
2. For an audio file that contains VBR data, allocates memory for the packet descriptions array. Calculates the memory needed based on the number of audio data packets to be read on each invocation of the playback callback. See Setting Buffer Size and Number of Packets to Read (page 53).
3. For an audio file that contains CBR data, such as linear PCM, the audio queue does not use a packet descriptions array.

Set a Magic Cookie for a Playback Audio Queue

Some compressed audio formats, such as MPEG 4 AAC, make use of structures to contain audio metadata. These structures are called **magic cookies**. When you play a file in such a format using Audio Queue Services, you get the magic cookie from the audio file and add it to the audio queue before you start playing.

Listing 3-14 shows how to obtain a magic cookie from a file and apply it to an audio queue. Your code would call this function before starting playback.

Listing 3-14 Setting a magic cookie for a playback audio queue

```
UInt32 cookieSize = sizeof (UInt32);                  // 1
bool couldNotGetProperty =                            // 2
    AudioFileGetPropertyInfo (                        // 3
        aqData.mAudioFile,                            // 4
        kAudioFilePropertyMagicCookieData,            // 5
        &cookieSize,                                  // 6
        NULL                                          // 7
    );

if (!couldNotGetProperty && cookieSize) {             // 8
    char* magicCookie =
        (char *) malloc (cookieSize);

    AudioFileGetProperty (                            // 9
        aqData.mAudioFile,                            // 10
        kAudioFilePropertyMagicCookieData,            // 11
        &cookieSize,                                  // 12
        magicCookie                                   // 13
    );

    AudioQueueSetProperty (                           // 14
        aqData.mQueue,                                // 15
        kAudioQueueProperty_MagicCookie,              // 16
        magicCookie,                                  // 17
        cookieSize                                    // 18
    );

    free (magicCookie);                               // 19
}
```

Here's how this code works:

1. Sets an estimated size for the magic cookie data.

2. Captures the result of the `AudioFileGetPropertyInfo` function. If successful, this function returns a value of `NoErr`, equivalent to Boolean `false`.
3. The `AudioFileGetPropertyInfo` function, declared in the `AudioFile.h` header file, gets the size of the value of a specified property. You use this to set the size of the variable that holds the property value.
4. An audio file object (of type `AudioFileID`) that represents the audio file you want to play.
5. The property ID representing an audio file's magic cookie data.
6. On input, an estimated size for the magic cookie data. On output, the actual size.
7. Uses `NULL` to indicate that you don't care about the read/write access for the property.
8. If the audio file does contain a magic cookie, allocate memory to hold it.
9. The `AudioFileGetProperty` function, declared in the `AudioFile.h` header file, gets the value of a specified property. In this case, it gets the audio file's magic cookie.
10. An audio file object (of type `AudioFileID`) that represents the audio file you want to play, and whose magic cookie you are getting.
11. The property ID representing the audio file's magic cookie data.
12. On input, the size of the `magicCookie` variable obtained using the `AudioFileGetPropertyInfo` function. On output, the actual size of the magic cookie in terms of the number of bytes written to the `magicCookie` variable.
13. On output, the audio file's magic cookie.
14. The `AudioQueueSetProperty` function sets a property in an audio queue. In this case, it sets a magic cookie for the audio queue, matching the magic cookie in the audio file to be played.
15. The audio queue that you want to set a magic cookie for.
16. The property ID representing an audio queue's magic cookie.
17. The magic cookie from the audio file that you want to play.
18. The size, in bytes, of the magic cookie.
19. Releases the memory that was allocated for the magic cookie.

Allocate and Prime Audio Queue Buffers

You now ask the audio queue that you've created (in Create a Playback Audio Queue (page 52)) to prepare a set of audio queue buffers. Listing 3-15 demonstrates how to do this.

Listing 3-15 Allocating and priming audio queue buffers for playback

```
aqData.mCurrentPacket = 0;                                  // 1

for (int i = 0; i < kNumberBuffers; ++i) {                  // 2
    AudioQueueAllocateBuffer (                              // 3
        aqData.mQueue,                                      // 4
        aqData.bufferByteSize,                              // 5
        &aqData.mBuffers[i]                                 // 6
    );

    HandleOutputBuffer (                                    // 7
        &aqData,                                            // 8
        aqData.mQueue,                                      // 9
        aqData.mBuffers[i]                                  // 10
    );
}
```

Here's how this code works:

1. Sets the packet index to `0`, so that when the audio queue callback starts filling buffers (step 7) it starts at the beginning of the audio file.
2. Allocates and primes a set of audio queue buffers. (You set this number, `kNumberBuffers`, to `3` in Define a Custom Structure to Manage State (page 41).)
3. The `AudioQueueAllocateBuffer` function creates an audio queue buffer by allocating memory for it.
4. The audio queue that is allocating the audio queue buffer.
5. The size, in bytes, for the new audio queue buffer.
6. On output, adds the new audio queue buffer to the `mBuffers` array in the custom structure.
7. The `HandleOutputBuffer` function is the playback audio queue callback you wrote. See Write a Playback Audio Queue Callback (page 43).
8. The custom structure for the audio queue.
9. The audio queue whose callback you're invoking.
10. The audio queue buffer that you're passing to the audio queue callback.

Set an Audio Queue's Playback Gain

Before you tell an audio queue to begin playing, you set its gain by way of the audio queue parameter mechanism. Listing 3-16 shows how to do this. For more on the parameter mechanism, see Audio Queue Parameters (page 21).

Listing 3-16 Setting an audio queue's playback gain

```
Float32 gain = 1.0;                                       // 1
    // Optionally, allow user to override gain setting here
AudioQueueSetParameter (                                  // 2
    aqData.mQueue,                                        // 3
    kAudioQueueParam_Volume,                              // 4
    gain                                                  // 5
);
```

Here's how this code works:

1. Sets a gain to use with the audio queue, between `0` (for silence) and `1` (for unity gain).
2. The `AudioQueueSetParameter` function sets the value of a parameter for an audio queue.
3. The audio queue that you are setting a parameter on.
4. The ID of the parameter you are setting. The `kAudioQueueParam_Volume` constant lets you set an audio queue's gain.
5. The gain setting that you are applying to the audio queue.

Start and Run an Audio Queue

All of the preceding code has led up to the process of playing a file. This includes starting an audio queue and maintaining a run loop while a file is playing, as shown in Listing 3-17.

Listing 3-17 Starting and running an audio queue

```
aqData.mIsRunning = true;                         // 1

AudioQueueStart (                                 // 2
    aqData.mQueue,                                // 3
    NULL                                          // 4
```

```
);

do {                                                // 5
    CFRunLoopRunInMode (                            // 6
        kCFRunLoopDefaultMode,                      // 7
        0.25,                                       // 8
        false                                       // 9
    );
} while (aqData.mIsRunning);

CFRunLoopRunInMode (                                // 10
    kCFRunLoopDefaultMode,
    1,
    false
);
```

Here's how this code works:

1. Sets a flag in the custom structure to indicate that the audio queue is running.
2. The `AudioQueueStart` function starts the audio queue, on its own thread.
3. The audio queue to start.
4. Uses `NULL` to indicate that the audio queue should start playing immediately.
5. Polls the custom structure's `mIsRunning` field regularly to check if the audio queue has stopped.
6. The `CFRunLoopRunInMode` function runs the run loop that contains the audio queue's thread.
7. Uses the default mode for the run loop.
8. Sets the run loop's running time to `0.25` seconds.
9. Uses `false` to indicate that the run loop should continue for the full time specified.
10. After the audio queue has stopped, runs the run loop a bit longer to ensure that the audio queue buffer currently playing has time to finish.

Clean Up After Playing

When you're finished playing a file, dispose of the audio queue, close the audio file, and free any remaining resources. Listing 3-18 illustrates these steps.

Listing 3-18 Cleaning up after playing an audio file

```
AudioQueueDispose (                             // 1
    aqData.mQueue,                              // 2
    true                                        // 3
);

AudioFileClose (aqData.mAudioFile);             // 4

free (aqData.mPacketDescs);                     // 5
```

Here's how this code works:

1. The `AudioQueueDispose` function disposes of the audio queue and all of its resources, including its buffers.
2. The audio queue you want to dispose of.
3. Use `true` to dispose of the audio queue synchronously.
4. Closes the audio file that was played. The `AudioFileClose` function is declared in the `AudioFile.h` header file.
5. Releases the memory that was used to hold the packet descriptions.

Document Revision History

This table describes the changes to *Audio Queue Services Programming Guide*.

Date	Notes
2013-12-19	Corrected code listings.
2010-07-09	Minor changes.
2007-10-31	New document that describes how to record and play audio using Audio Queue Services.

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