Audio Session Programming Guide

Audio & Video: Audio

2010-07-09

Contents

Figures, Tables, and Listings

About Configuring Audio Behavior

iOS handles audio behavior at the application, interapplication, and device levels using the notion of *audio sessions*. Using the audio session API, you implement answers to questions such as these:

- Should your application's audio be silenced by the Ring/Silent switch? The answer to this question is "Yes" if audio is not essential for effective use of your app. An example might be a note-taking app or a game that lets users discretely use the app in a meeting—without its sound disturbing others. A pronouncing dictionary, on the other hand, should ignore the Ring/Silent switch and always play—the central purpose of the app requires sound.
- Should iPod audio continue when your audio starts? If your app is a virtual piano that lets users play along to songs from their iPod libraries, the answer to this question is certainly "Yes." On the other hand, iPod audio should stop when your app starts, for example, if your app plays streaming Internet radio.

Users may plug in or unplug headsets, phone calls may arrive, and alarms may sound. Indeed, the audio environment on an iOS device is quite complex. iOS does the heavy lifting, while you employ audio session APIs to specify configuration and to respond gracefully to system requests, using very little code.

An Audio Session Encapsulates a Set of Behaviors

An **audio session** is the intermediary between your application and iOS for configuring audio behavior. Upon launch, your application automatically gets a singleton audio session. The behavior you specify should meet user expectations as described in "Using Sound" in *iPhone Human Interface Guidelines*.

Relevant Chapters: "Audio Session Basics" (page 13) and "Configuring the Audio Session" (page 19).

Categories Express Audio Roles

The primary mechanism for expressing audio intentions is to set the audio session category. A **category** is a key that identifies a set of audio behaviors. By setting the category, you indicate whether your audio should continue when the screen locks, whether you want iPod audio to continue playing along with your audio, and so on.

Six audio session categories, along with a set of override and modifier switches, let you customize audio behavior according to your application's personality or role. Various categories support playback, recording, playback along with recording, and offline audio processing. When the system knows your app's audio role, it affords you appropriate access to hardware resources. The system also ensures that other audio on the device behaves in a way that works for your application; for example, if you need iPod audio to be silenced, it is.

Relevant Chapters: "Configuring the Audio Session" (page 19) and "Audio Session Categories" (page 63).

Delegates Support Interruption Handling

An **audio interruption** is the deactivation of your application's audio session—which immediately stops your audio. Interruptions happen when a competing audio session from a built-in application activates and that session is not categorized by the system to mix with yours. After your session goes inactive, the system sends a "you were interrupted" message which you can respond to by saving state, updating the user interface, and so on.

To handle interruptions, implement Objective-C interruption delegate methods provided by the AV Foundation framework. Write your `beginInterruption` and `endInterruption` methods to ensure the minimum possible disruption, and the most graceful possible recovery, from the perspective of the user.

Relevant Chapters: "Handling Audio Interruptions" (page 25).

Callbacks Support Audio Route Change Handling

Users have particular expectations when they initiate an **audio route change** by docking or undocking a device, or by plugging in or unplugging a headset. "Using Sound" in *iPhone Human Interface Guidelines* describes these expectations and provides guidelines on how to meet them. Handle route changes by writing a C callback function and registering it with your audio session.

Relevant Chapters: "Handling Audio Hardware Route Changes" (page 31).

Properties Support Advanced Features

You can fine-tune an audio session category in a variety of ways. Depending on the category, you can:

- Allow other audio (such as from the iPod) to mix with yours when a category normally disallows it
- Change the audio output route from the receiver to the speaker
- Allow Bluetooth audio input
- Specify that other audio should reduce in volume ("duck") when your audio plays

You also use properties to optimize your application for device hardware at runtime. This lets your code adapt to the characteristics of the device it's running on, as well as to changes in hardware capabilities initiated by the user (such as by plugging in a headset) as your application runs.

To use properties, employ the C-based Audio Session Services API from the Audio Toolbox framework.

Relevant Chapters: "Configuring the Audio Session" (page 19), "Optimizing for Device Hardware" (page 35), and "Working with Movies and iPod Music" (page 39)

How to Use This Document

All iOS developers who use the AV Foundation framework, Audio Queue Services, OpenAL, or the I/O audio unit should read this document in its entirety. Other iOS developers should read at least this and the next chapter, "Audio Session Basics" (page 13). If you are already up-to-speed on audio session concepts, go straight to the "Audio Session Cookbook" (page 43) chapter.

An appendix, "Audio Session Categories" (page 63), provides details on the behavior of each iOS audio session category.

Prerequisites

Before reading this document, you should be familiar with Cocoa Touch development as introduced in *iOS Application Programming Guide* and with the basics of Core Audio as described in that document and in *Core Audio Overview*. Because audio sessions bear on practical end-user scenarios involving the various switches, buttons, and connectors on a device, you should also be familiar with using iPhone.

See Also

You may find the following resources helpful as you learn about using audio sessions:

- *Audio Session Services Reference*, the C-language, companion API reference to this document.
- *AVAudioSession Class Reference*, which describes the Objective-C interface for configuring and using this technology.
- *Core Audio Overview*, which describes the architecture and design principles of Core Audio in iOS and Mac OS X.
- *iOS Application Programming Guide*, an essential introduction to programming Cocoa Touch applications.
- *iPhone Human Interface Guidelines*, which includes user interface best practices for audio applications. In particular, see the section "Using Sound" in *iPhone Human Interface Guidelines*.
- *AddMusic*, a sample code project that demonstrates use of an audio session object in the context of a playback application. This sample also demonstrates coordination between application audio and iPod audio.

Availability

The Audio Session APIs are specialized for the needs of iOS devices, and so are available only in iOS. Basic audio session support was available in iOS 2.0. Some of the features described in this guide require iOS 3.1

Audio Session Basics

Read this chapter to learn which iOS development problems audio sessions solve and to make first acquaintance with audio session features.

Why Does iOS Need to Manage Audio?

On your morning commute you unlock your iPhone and start listening to a new episode of a podcast, which plays back through the built-in speaker. As your seatmate frowns, you quickly plug in your headset and the Podcast continues, its output now rerouted and at the volume you last used for the headset. Maybe you start a sudoku, which plays its own sound effects—that mix with the podcast output. A few seconds later the podcast fades to silence, an alarm sounds, and an alert appears, reminding you of a birthday. You dismiss the alert. The podcast fades back in and resumes where it left off. Sounds in your sudoku game resume working.

You might do all this in the space of a minute—and without touching any audio settings. The remarkable simplicity of the audio user experience on iPhone belies an underlying complexity greater than that on a Mac Pro. The infrastructure that makes the simplicity possible is exposed to your application through an audio session object.

An audio session lets you provide a seamless audio experience in your application. In fact, any iOS application that uses the AV Foundation framework, Audio Queue Services, OpenAL, or the I/O audio unit must use the audio session programming interface to meet Apple's recommendations as laid out in *iPhone Human Interface Guidelines*.

What Is an Audio Session?

An **audio session** is the intermediary between your application and iOS for configuring audio behavior. Upon launch, your application automatically gets a singleton audio session. You configure it to express your application's audio intentions. For example:

- Do you intend to mix your application's sounds with those from other applications (such as the iPod), or do you intend to silence other audio?
- How do you want your application to respond to an audio interruption, such as a Clock alarm?
- How should your application respond when a user plugs in or unplugs a headset?

Audio session configuration influences all audio activity while your application is running, except for user-interface sound effects that you play. You can query the audio session to discover hardware characteristics of the device your application is on—characteristics such as channel count, sample rate, and availability of audio input. These can vary by device and can change due to user actions while your application runs.

You can explicitly activate and deactivate your audio session. For application sound to play, or for recording to work, your audio session must be active. The system can also deactivate your audio session—which it does, for example, when a phone call arrives or an alarm sounds. Such a deactivation is called an **interruption**. The audio session APIs provide ways to respond to and recover from interruptions.

What Is an Audio Session Category?

An audio session **category** is a key that identifies a set of audio behaviors for your application. By setting a category, you indicate your audio intentions to the system—such as whether your audio should continue when the screen locks. The six audio session categories in iOS, along with a set of override and modifier switches, let you customize your app's audio behavior.

Each audio session category specifies a particular pattern of "yes" and "no" for each of the following behaviors, as detailed in "Audio Session Categories" (page 63):

- **Allows mixing**: if yes, audio from other applications (such as the iPod) can continue playing when your application plays sound.
- **Silenced by the Ring/Silent switch and by screen locking**: if yes, your audio is silenced when the user moves the Ring/Silent switch to silent and when the screen locks.
- **Supports audio input**: if yes, application audio input, such as for recording, is allowed.
- **Supports audio output**: if yes, application audio output, such as for playback, is allowed.

Most applications need only set the category once, at launch. That said, you can change the category as often as you need to, and can do so whether your session is active or inactive. If your session is inactive, your category request is sent when you activate your session. If your session is already active, your category request is sent immediately.

Audio Session Default Behavior—What You Get for Free

An audio session comes with some default behavior. Specifically:

- Playback is enabled and recording is disabled.
- When the user moves the Ring/Silent switch to the "silent" position, your audio is silenced.
- When the user presses the Sleep/Wake button to lock the screen, or when the Auto-Lock period expires, your audio is silenced.
- When your audio starts, other audio on the device—such as iPod audio that was already playing—is silenced.

This collection of behavior is encapsulated in, and named by, the `AVAudioSessionCategorySoloAmbient` audio session category—the default category.

Your audio session is automatically activated on application launch. This allows you to play (or record, if you specify one of the categories that supports audio input). However, relying on the default activation is a risky state for your application. For example, if an iPhone rings and the user ignores the call—leaving your application running—your audio may no longer play, depending on which playback technology you're using. The next section describes some strategies for dealing with such issues, and "Handling Audio Interruptions" (page 25) goes into depth on the topic.

You can take advantage of default behavior as you're bringing your application to life during development. However, the only times you can safely ignore the audio session for a shipping application are these:

- Your application uses System Sound Services, exclusively, for audio.

  System Sound Services is the iOS technology for playing user-interface sound effects and for invoking vibration. It is unsuitable for other purposes. (See *System Sound Services Reference* and the *Audio UI Sounds (SysSound)* sample code project.)
- Your application uses no audio at all.

In all other cases, do not ship your application with the default audio session. You may elect to use the default category, but explicitly using and managing an audio session that employs the "solo ambient" category provides different behavior than using it by default, as described next.

Why a Default Audio Session Usually Isn't What You Want

If you don't initialize, configure, and explicitly use your audio session, your application cannot respond to interruptions or audio hardware route changes. Moreover, your application would have no voice in OS decisions about audio mixing between applications.

Note: Default audio session behavior changed between iOS 2.1 and 2.2. Starting in iOS 2.2, audio, by default, is silenced when the user moves the Ring/Silent switch to the "silent" position or when the screen locks. To ensure that your audio keeps playing or recording in these circumstances, assign an appropriate category to your audio session. For details, see "Audio Session Categories" (page 63) and "Setting the Category" (page 44).

Here are some scenarios that clarify audio session default behavior and how you can change it:

- Scenario 1. You write an audio book application. A user begins listening to *The Merchant of Venice*. As soon as Lord Bassanio enters, Auto-Lock times out, the screen locks, and your audio goes silent.

  To ensure that audio continues upon screen locking, configure your audio session to support that. For playback that continues when the screen is locked, use the `AVAudioSessionCategoryPlayback` category.
- Scenario 2. You write a first-person shooter game that uses OpenAL-based sound effects. You also provide a background soundtrack but include an option for the user to turn it off, instead using iPod Library Access to play a song from their iPod library. The user starts up their favorite killing song. The first time they fire a photon torpedo at an enemy ship, the iPod music stops.

  To ensure that iPod music is not interrupted, configure your audio session to allow mixing. Use the `AVAudioSessionCategoryAmbient` category; or modify the playback category to support mixing.
- Scenario 3. You write a streaming radio application that uses Audio Queue Services for playback. While a user is listening, a phone call arrives and stops your sound, as expected. The user chooses to ignore the call and dismisses the alert. The user taps Play again to resume the music stream, but nothing happens. To resume playback, the user must quit your application and restart it.

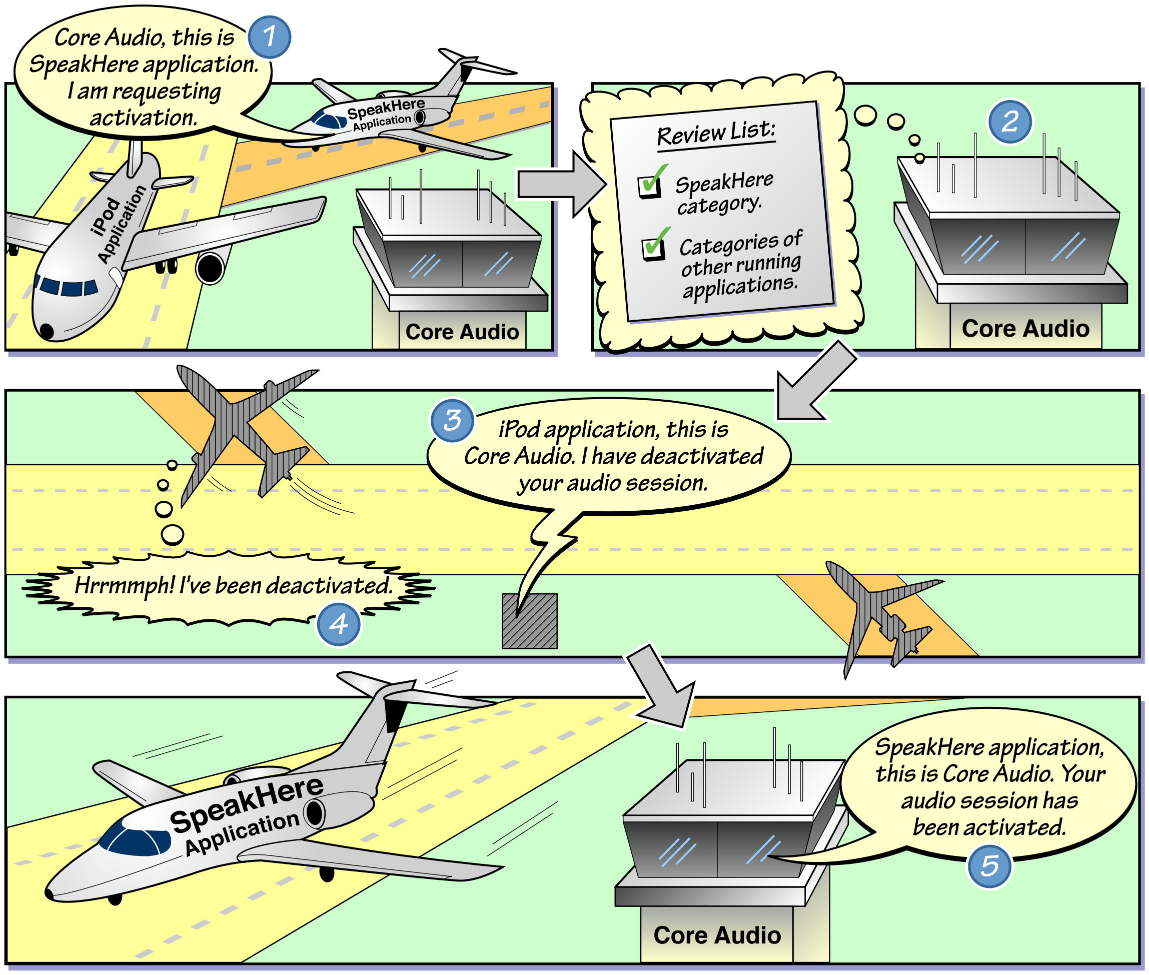
To handle the interruption of an audio queue gracefully, implement delegate methods or write an audio session callback function to allow your application to continue playing automatically or to allow the user to manually resume playing. See "Responding to Audio Session Interruptions" (page 50).

How the System Resolves Competing Audio Demands

As your iOS application launches, built-in applications (Messages, iPod, Safari, the phone) may be running in the background. Each of these may want to produce audio: a text message arrives, a podcast you started 10 minutes ago continues playing, and so on.

If you think of an iPhone as an airport, with applications represented as taxiing planes, the system serves as a sort of control tower. Your application can make audio requests and state its desired priority, but final authority over what happens "on the tarmac" comes from the system. You communicate with the "control tower" using the audio session. Figure 1-1 illustrates a typical scenario—your application wanting to use audio while the iPod is already playing. In this scenario, your application, in effect, interrupts the iPod.

Figure 1-1 Core Audio manages competing audio demands


1
Core Audio, this is SpeakHere application. I am requesting activation.
SpeakHere Application
iPod Application
Core Audio
Review List:
SpeakHere category.
Categories of other running applications.
2
Core Audio
3
iPod application, this is Core Audio. I have deactivated your audio session.
Hrrmmph! I've been deactivated.
4
SpeakHere Application
Core Audio
SpeakHere application, this is Core Audio. Your audio session has been activated.
5

In step 1 in the figure, your application requests activation of its audio session. You'd make such a request, for example, on application launch, or perhaps in response to a user tapping the Play button in an audio recording and playback application. In step 2, Core Audio considers the activation request. Specifically, it considers the category you've assigned to your audio session. In Figure 1-1, the SpeakHere application uses a category that requires other audio to be silenced.

In steps 3 and 4 in the figure, Core Audio deactivates the iPod application's audio session, stopping its audio playback. Finally, in step 5, Core Audio activates the SpeakHere application's audio session and playback begins.

Core Audio manages competing audio demands by having final authority to activate or deactivate any of the audio sessions present on a device. In deciding, it follows the inviolable rule that "the phone always wins." No application, no matter how vehemently it demands priority, can trump the phone. When a call arrives, the user gets notified and your application is interrupted—no matter what audio operation you have in progress and no matter what category you have set.

To ensure that your audio is not disrupted by a phone call or a Clock alarm, the user must turn on Airplane Mode. This highlights another inviolable rule: the user, not your application, is in control of the device. For example, there is no programmatic way to silence a Clock alarm. To prevent an alarm from ruining a recording, the user must turn off scheduled alarms. Similarly, there is no way to programmatically set hardware playback volume. The user is always in control of the hardware volume using the volume buttons on the side of the device.

The Two Audio Session APIs

iOS offers two APIs for working with the audio session object, each with its own advantages:

- The **AVAudioSession class**, described in *AVAudioSession Class Reference*, provides a convenient, Objective-C interface that works well with the other Objective-C code in your application. This API provides access to a core set of audio session features.

  `AVAudioSession` has two key advantages. When you use it to obtain the shared instance of the audio session, the session is implicitly initialized—there is no separate initialization to perform as there is when using the C API. Second, you can take advantage of simple delegate methods for handling audio interruptions and changes to the hardware configuration such as sample rate and channel count. You can use these delegate methods no matter which audio technology you are using for playback, recording, or processing.

- **Audio Session Services**, described in *Audio Session Services Reference*, is a full-featured C API that provides access to all basic and advanced features of the audio session.

  You need this API for handling audio hardware route changes as well as for making modifications to the standard behavior of audio session categories. For example, the `kAudioSessionProperty_OverrideAudioRoute` property lets you switch playback from the receiver to the speaker when using the "play and record" audio session category. This API also provides a callback mechanism for handling interruptions.

You can mix and match calls to the two audio session APIs—they are completely compatible with each other. The rest of this document goes into depth on using the various features of these APIs.

Developing with the Audio Session APIs

When you add audio session support to your application, you can run your app in the Simulator or on a device. However, the Simulator does not simulate audio session behavior and does not have access to the hardware features of a device. When running in the simulator, you cannot:

- Invoke an interruption
- Change the setting of the Ring/Silent switch
- Simulate screen lock
- Simulate the plugging in or unplugging of a headset
- Query audio route information or test audio session category behavior
- Test audio mixing behavior—that is, playing your audio along with audio from another application (such as the iPod)

To test the behavior of your audio session code, you need to run on a device. For some development tips when working with the Simulator, see "Running Your App in the Simulator" (page 61).

Configuring the Audio Session

Configuring the audio session establishes basic audio behavior for your application. Most importantly, you set the audio session category according to what your app does and how you want it to interact with the device and the system. You can change the category, if needed, as your application runs.

Prerequisite to configuration, though, is initialization, described first.

Initializing the Audio Session

The system gives you an audio session object upon launch of your application. Before working with the session, you must initialize it. How you handle initialization depends primarily on how you want to handle audio interruptions:

- If using the AV Foundation framework for managing interruptions (explained in "Handling Interruptions Using Delegate Methods" (page 27)), take advantage of the implicit initialization that occurs when you obtain a reference to the `AVAudioSession` object, as shown here:

  ```
  // implicitly initializes the audio session
  AVAudioSession session = [AVAudioSession sharedInstance];
  ```

  The `session` variable now represents the initialized audio session and can be used immediately. Apple recommends using implicit initialization when you handle audio interruptions with the `AVAudioSession` class's interruption delegate methods, or with the delegate protocols of the `AVAudioPlayer` and `AVAudioRecorder` classes.

- Alternatively, you can write a C callback function to handle audio interruptions (explained in "Handling Interruptions Using a Callback Function" (page 28)). You must then attach that code to the audio session by using explicit initialization with the `AudioSessionInitialize` function. Typically, you'd perform explicit audio session initialization in your application's main controller class, inside that class's initialization method. "Initializing an Audio Session" (page 43) provides a code example.

Activating and Deactivating the Audio Session

The system activates your audio session on application launch. Even so, Apple recommends that you explicitly activate your session—typically as part of your application's `viewDidLoad` method. This gives you an opportunity to test whether or not activation succeeded. Likewise, when changing the audio session's active/inactive state, check to ensure that the call is successful. Write your code to gracefully handle the system refusing to activate your session.

The system deactivates your audio session for a Clock or Calendar alarm or incoming phone call. When the user dismisses the alarm, or chooses to ignore a phone call, the system allows your session to become active again. Do reactivate it, upon interruption end, to ensure that your audio works. For code examples, see "Activating and Deactivating the Audio Session" (page 44).

In the specific case of playing or recording audio with an `AVAudioPlayer` or `AVAudioRecorder` object, the system takes care of audio session reactivation.

Most applications never need to explicitly deactivate their audio session. A possible exception is a recording application. An active session should have the "recording" category only when the application is recording. When recording stops, you can deactivate the session to allow other sounds, such as incoming text message alerts, to play. Only deactivate the session, though, if your application has no sounds to play. If you provide playback of recorded audio, for example, change the category from recording to playback instead of deactivating the session.

Choosing the Best Category

iOS has six audio session categories:

- Three for playback
- One for recording
- One that supports playback and recording—that need not occur simultaneously
- One for offline audio processing

To pick the best category, consider a few factors:

- The best category is the one that most closely supports your audio needs. Do you want to play audio, record audio, do both, or just perform offline audio processing?
- Is audio that you play essential or peripheral to using your application? If essential, the best category is one that supports playback with the Ring/Silent switch set to silent. If peripheral, pick a category that goes silent with the Ring/Silent switch set to silent.
- Is other audio (such as iPod audio) playing when the user launches your application? Checking during launch enables you to branch. For example, a game application could choose a category configuration that allows iPod audio to continue if it's already playing, or choose a different category configuration to support a built-in app soundtrack otherwise.

The following list describes the categories. The first category allows other audio to continue playing; the remaining categories indicate that you want other audio to stop when your session becomes active. However, you can customize the nonmixing "playback" and "play and record" categories to allow mixing, as described in "Fine-Tuning the Category" (page 22).

- `AVAudioSessionCategoryAmbient` or the equivalent `kAudioSessionCategory_AmbientSound`—Use this category for an application that plays sounds that add polish or interest but are not essential to the application's use. Using this category, your audio is silenced by the Ring/Silent switch and when the screen locks.

  This category allows audio from the iPod, Safari, and other built-in applications to play while your application is playing audio. You could, for example, use this category for an application that provides a virtual musical instrument that a user plays along to iPod audio.

- `AVAudioSessionCategorySoloAmbient` or the equivalent `kAudioSessionCategory_SoloAmbientSound`—Use this category for an application whose audio you want silenced when the user switches the Ring/Silent switch to the "silent" position and when the screen locks. This is the default category. It differs from the `AVAudioSessionCategoryAmbient` category only in that it silences other audio.
- `AVAudioSessionCategoryPlayback` or the equivalent `kAudioSessionCategory_MediaPlayback`—Use this category for an application whose audio playback is of primary importance. Your audio plays even with the screen locked and with the Ring/Silent switch set to silent.

> **Note:** If you choose an audio session category that allows audio to keep playing when the screen locks, you should normally *not* disable the system's sleep timer. The sleep timer ensures that the screen goes dark after a user-specified interval, saving battery power.

- `AVAudioSessionCategoryRecord` or the equivalent `kAudioSessionCategory_RecordAudio`—Use this category for recording audio. All playback on the phone—except for ringtones and Clock and Calendar alarms—is silenced.
- `AVAudioSessionCategoryPlayAndRecord` or the equivalent `kAudioSessionCategory_PlayAndRecord`—Use this category for an application that inputs and outputs audio. The input and output need not occur simultaneously, but can if needed. This is the category to use for audio chat applications.
- `AVAudioSessionCategoryAudioProcessing` or the equivalent `kAudioSessionCategory_AudioProcessing`—Use this category when performing offline audio processing and not playing or recording.

The precise behaviors associated with each category are not under your application's control, but rather are set by the operating system. Apple may refine category behavior in future versions of iOS. Your best strategy is to pick the category that most accurately describes your intentions for the audio behavior you want. The appendix, "Audio Session Categories" (page 63), describes the behavior details for each category.

You may want to check, during application launch, whether other audio is playing and then choose your category based on that. During launch—such as within the `viewDidLoad` method—check the value of the `kAudioSessionProperty_OtherAudioIsPlaying` property.

How Categories Affect Encoding and Decoding

In iOS, you can encode uncompressed audio to a variety of compressed formats. You can also decode these formats for playback. The hardware-assisted codecs on a device consume less power than do the software codecs. In addition, the hardware codecs perform work that would otherwise be done by the main processor—thereby freeing CPU cycles for other tasks. For best performance and battery life, use the hardware-assisted codecs, if possible, when working with compressed audio formats. This can be particularly important for graphics-intensive applications such as games; using software audio decoding may limit your application's maximum video frame rate.

A device's hardware-assisted codecs are available to your application only if you configure your audio session category to silence other audio (such as iPod audio). In other words, to gain access to hardware-assisted audio encoding or decoding, you must claim exclusive rights to audio playback. Likewise, to use hardware-assisted encoding to a compressed audio format, your application must claim exclusive rights to audio input.

The following categories allow use of hardware-assisted audio encoding and decoding:

- `AVAudioSessionCategorySoloAmbient` or the equivalent `kAudioSessionCategory_SoloAmbientSound`
- `AVAudioSessionCategoryPlayback` or the equivalent `kAudioSessionCategory_MediaPlayback`
- `AVAudioSessionCategoryPlayAndRecord` or the equivalent `kAudioSessionCategory_PlayAndRecord`
- `AVAudioSessionCategoryAudioProcessing` or the equivalent `kAudioSessionCategory_AudioProcessing`

If you override the non-mixing nature of a playback category, as described in "Fine-Tuning the Category" (page 22), you cannot use hardware-assisted codecs. If you are performing offline audio conversion using a hardware-assisted codec, and don't need to simultaneously play audio, use the `AVAudioSessionCategoryAudioProcessing` (or the equivalent `kAudioSessionCategory_AudioProcessing`) category.

Setting and Changing the Audio Session Category

For most iOS applications, setting the audio session category at launch—and never changing it—works well. This provides the best user experience because the device's audio behavior remains consistent as your application runs. The main exception to this guideline is that an audio recording application should change categories depending on its state.

For recording, use the `AVAudioSessionCategoryRecord` (or the equivalent `kAudioSessionCategory_RecordAudio`) category. This ensures that recorded audio is not compromised by device sounds such as from incoming text messages. When finished recording—such as when the user taps Stop—change the category to the appropriate playback category depending on the needs of your application. Any of the playback categories allow text message alert sounds to be heard.

These same considerations apply for audio measurement applications that use audio input but don't necessarily record, such as an audio spectrum analyzer.

For code examples that show how to set the audio session category, see "Setting the Category" (page 44) in the "Audio Session Cookbook" chapter.

Fine-Tuning the Category

You can fine-tune an audio session category in a variety of ways. Depending on the category, you can:

- Allow other audio (such as from the iPod) to mix with yours when a category normally disallows it

- Change the audio output route from the receiver to the speaker
- Allow Bluetooth audio input
- Specify that other audio should reduce in volume ("duck") when your audio plays

You can override the non-mixing characteristic of the `AVAudioSessionCategoryPlayback` (or the equivalent `kAudioSessionCategory_MediaPlayback`), and `AVAudioSessionCategoryPlayAndRecord` (or the equivalent`kAudioSessionCategory_PlayAndRecord`) categories. To perform the override, apply the `kAudioSessionProperty_OverrideCategoryMixWithOthers` property to the audio session. See "Modifying Playback Mixing Behavior" (page 46) for a code example.

When you modify a playback category to allow other audio (such as iPod audio) to mix with yours, you cannot use hardware audio decoding of compressed formats.

You can programmatically influence the audio output route. When using the `AVAudioSessionCategoryPlayAndRecord` (or the equivalent`kAudioSessionCategory_PlayAndRecord`) category, audio normally goes to the receiver (the small speaker you hold to your ear when on a phone call). You can redirect audio to the speaker at the bottom of the phone by using a category routing override. For code examples, see "Redirecting Output Audio" (page 49).

Starting in iOS 3.1, you can configure the recording categories to allow input from a paired Bluetooth device that supports HFP. For a code example, see "Supporting Bluetooth Audio Input" (page 50).

Finally, you can enhance a category to automatically lower the volume of other audio when your audio is playing. This could be used, for example, in an exercise application. Say the user is exercising along to their iPod when your application wants to overlay a verbal message—for instance, "You've been rowing for 10 minutes." To ensure that the message from your application is intelligible, apply the `kAudioSessionProperty_OtherMixableAudioShouldDuck` property to the audio session. When ducking takes place, all other audio on the device—apart from phone audio—lowers in volume.

Handling Audio Interruptions

Adding audio session code to handle interruptions ensures that your application's audio continues behaving gracefully when a phone call arrives or a Clock or Calendar alarm sounds.

An **audio interruption** is the deactivation of your application's audio session—which immediately stops or pauses your audio, depending on which technology you are using. Interruptions happen when a competing audio session from a built-in application activates and that session is not categorized by the system to mix with yours. After your session goes inactive, the system sends a "you were interrupted" message which you can respond to by saving state, updating the user interface, and so on.

Your application may get shut down following an interruption. This happens when a user decides to accept a phone call. If a user instead elects to ignore a call, or dismisses an alarm, the system issues an interruption-ended message and your application continues running. For your audio to resume, your audio session must be reactivated.

Audio Interruption Handling Techniques

There are two alternative approaches for responding to interruptions:

- Implement Objective-C interruption delegate methods provided by the AV Foundation framework. In most cases, this approach is simpler and fits better with the rest of your application code.
- Write a C-based interruption callback function as declared in Audio Session Services. This option requires you to register your callback with the audio session by using an explicit audio session initialization call.

With either approach, what you do within your interruption code depends a great deal on the audio technology you are using and on what you are using it for—playback, recording, audio format conversion, reading streamed audio packets, and so on. Generally speaking, you need to ensure the minimum possible disruption, and the most graceful possible recovery, from the perspective of the user.

Table 3-1 summarizes the steps that need to happen during an interruption. When using the `AVAudioPlayer` or `AVAudioRecorder` objects, some of these steps are handled automatically by the system.

Table 3-1 What needs to happen during an audio session interruption

After interruption starts	■ Check whether resumption of audio process is supported ■ Save state and context ■ Update user interface
After interruption ends	■ Restore state and context ■ Reactivate audio session ■ Update user interface

Table 3-2 briefly summarizes how to handle audio interruptions according to technology. The rest of this chapter provides details.

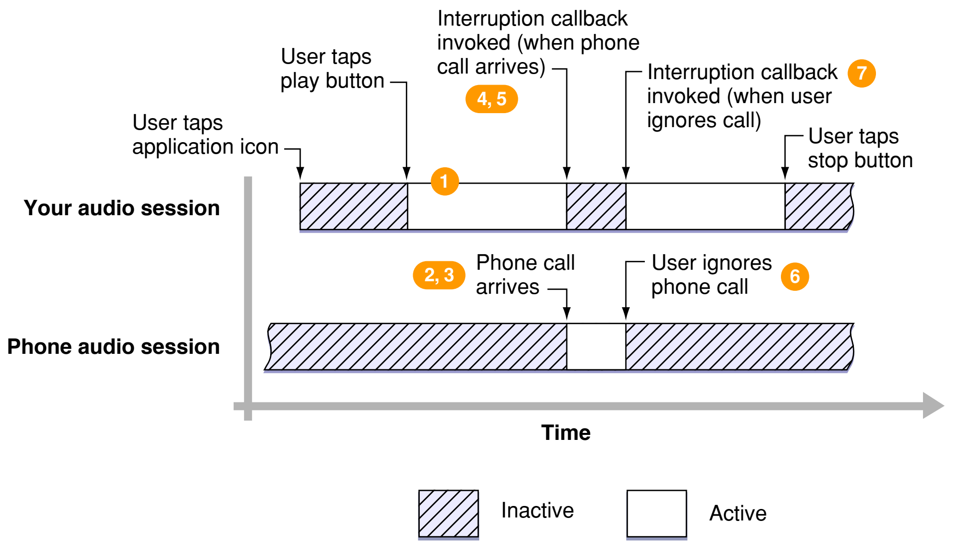
Table 3-2 Audio interruption handling techniques according to audio technology

Audio technology	How interruptions work
AV Foundation framework	The `AVAudioPlayer` and `AVAudioRecorder` classes provide delegate methods for interruption start and end. Implement these methods to update your user interface and optionally, after interruption ends, to resume paused playback. The system automatically pauses playback or recording upon interruption, and reactivates your audio session when you resume playback or recording. If you want to save and restore playback position between application launches, save playback position on interruption as well as on application quit.
Audio Queue Services, I/O audio unit	These technologies put your application in control of handling interruptions. You are responsible for saving playback or recording position and reactivating your audio session after interruption ends. As explained in "Responding to Audio Session Interruptions" (page 50), implement the `AVAudioSession` interruption delegate methods or write an interruption listener callback function. If using these technologies for hardware-assisted encoding to AAC, refer to "Hardware-Assisted Codecs and Audio Interruptions" (page 29).
OpenAL	When using OpenAL for playback, implement the `AVAudioSession` interruption delegate methods or write an interruption listener callback function—as when using Audio Queue Services. However, the delegate or callback must additionally manage the OpenAL context. See "OpenAL and Audio Interruptions" (page 29).
System Sound Services	Sounds played using System Sound Services go silent when an interruption starts. They can automatically be used again if the interruption ends. Applications cannot influence the interruption behavior for sounds that use this playback technology.

The Interruption Life Cycle

Figure 3-1 illustrates the sequence of events before, during, and after an audio session interruption for a playback application.

Figure 3-1 An audio session gets interrupted



An interruption event—in this example, the arrival of a phone call—proceeds as follows. The numbered steps correspond to the numbers in the figure.

1. Your application is active, playing back audio.
2. A phone call arrives. The system activates the phone application's audio session.
3. The system deactivates your audio session. At this point, playback in your application has stopped.
4. The system invokes your interruption listener callback function (or calls your interruption-started delegate method), indicating that your session has been deactivated.
5. Your callback (or delegate method) takes appropriate action. For example, it could update the user interface and save the information needed to resume playback at the point where it stopped.
6. If the user dismisses the interruption—electing to ignore an incoming phone call in this case—the system invokes your callback or delegate method, indicating that the interruption has ended.
7. Your callback or delegate method takes action appropriate to the end of an interruption. For example, it updates the user interface, reactivates your audio session, and resumes playback.
8. (Not shown in the figure.) If, instead of dismissing the interruption at step (6), the user accepted a phone call, your application would have then quit. This is the other possible end of the interruption life cycle.

Handling Interruptions Using Delegate Methods

The `AVAudioSession` class provides delegate methods, described in *AVAudioSessionDelegate Protocol Reference*, for responding to interruptions. You can implement and use these methods to respond to interruptions no matter which audio technology you are are using:

- `beginInterruption` Called after your audio session is interrupted. Implement this method assuming that your application is about to be shut down—such as when a user accepts an incoming phone call. For example, if you want to let a user resume playback from the point of interruption on the next launch of your application, you would save the file identifier and the current file frame number.
- `endInterruption` Called after your audio session interruption has ended. If not using an `AVAudioPlayer` or `AVAudioRecorder` object, which automatically reactivate the audio session, you must explicitly reactivate the audio session.

What you do within these methods depends on the audio technology you're using and on what you're using it for. At a minimum, ensure that your internal state and user interface remain consistent and, if you get the `endInterruption` message, that your audio session is reactivated.

The `AVAudioRecorder` and `AVAudioPlayback` classes provide their own delegate methods for responding to audio interruptions, as listed here:

- `audioPlayerBeginInterruption:` Called when an audio session is interrupted during playback.
- `audioPlayerEndInterruption:` Called when a playback interruption ends.
- `audioRecorderBeginInterruption:` Called when the audio session is interrupted during a recording.
- `audioRecorderEndInterruption:` Called when a recording interruption ends.

The `AVAudioRecorder` and `AVAudioPlayback` classes automatically pause upon interruption. You need not save the recording or playback position unless you want that position to persist across launches of your application. In addition, the system automatically reactivates your audio session should you resume playback or recording after an interruption ends.

For a code example showing how to use an AV Foundation interruption delegate method, see "Handling Interruptions with the AVAudioPlayer Class" (page 51).

Handling Interruptions Using a Callback Function

If you prefer to use a C-language interruption listener callback function rather than Objective-C delegate methods, base your callback on the `AudioSessionInterruptionListener` callback prototype from Audio Session Services.

When the system invokes your callback, it sends two values: a reference to the data that you (optionally) specified when you initialized the session and a constant that indicates the interruption state. Listing 3-1 shows the declaration of the `AudioSessionInterruptionListener` callback.

Listing 3-1 The interruption callback declaration

```
typedef void (*AudioSessionInterruptionListener) (
                 void    *inClientData,
                 UInt32  inInterruptionState
             );
```

There are two interruption states:

- `kAudioSessionBeginInterruption` Your audio session has just been interrupted and made inactive. On getting this message, your application should assume it will be terminated—which does happen, say, if a user elects to take the phone call that is the source of the interruption. Write your interruption callback to perform any actions needed before your application gets shut down.
- `kAudioSessionEndInterruption` The interruption had ended. Your callback would get this constant if, for example, a user elects to ignore an incoming phone call. On getting this message, your callback can tell your application to reactivate its audio session and resume playback, or to otherwise return to an appropriate state. If a user takes the call, however, your application is then stopped. In this case, your callback is not invoked with this constant.

When using a callback, there are three parts to adding interruption support to your application:

1. Define methods that take appropriate action upon interruption-begin and interruption-end. Some of these actions—such as to stop recording—are ones you define anyway for your application. Others—such as to pause playback—may be ones that you invoke only from the callback. See "Defining Interruption Methods" (page 52).

2. Define a callback function, to be invoked upon interruption begin and end, that calls these methods as appropriate. See "Defining an Interruption Listener Callback Function" (page 53).

3. Register the callback function with your audio session. You do this as part of initializing the session. See "Initializing an Audio Session" (page 43).

OpenAL and Audio Interruptions

When using OpenAL for audio playback, implement `AVAudioSession` delegate methods or write an interruption listener callback function, as you do when using Audio Queue Services. However, your interruption code must additionally manage the OpenAL context. Upon interruption, set the OpenAL context to `NULL`. After interruption ends, set the context to its prior state.

Prior to iOS version 3.0, handling OpenAL interruptions was more involved. You had to save and then destroy the OpenAL context after getting an interruption, and then rebuild the context on interruption end. To make your code compatible with older versions of iOS, you can conditionalize it by testing the OS version at runtime.

For code examples showing how to manage the OpenAL context during an audio interruption, see "Responding to Interruptions When Using OpenAL" (page 54).

Hardware-Assisted Codecs and Audio Interruptions

Previous sections in this chapter explained how to handle interruptions to the two most common audio operations: playing back and recording. Another operation that requires interruption handling is offline audio format conversion. You have a bit more work to do in this case. Specifically, you must handle interruptions at the audio data buffer level.

By way of background, you can use a hardware assisted-codec—on certain devices—to encode linear PCM audio to AAC format. The codec is available on the iPhone 3GS and on the iPod touch (2nd generation), but not on older models. You use the codec as part of an audio converter object (of type `AudioConverterRef`), which in turn is part of an extended audio file object (of type `ExtAudioFileRef`). For information on these opaque types, refer to *Audio Converter Services Reference* and *Extended Audio File Services Reference*.

To handle an interruption during hardware-assisted encoding, take two things into account:

1. The codec may or may not be able to resume encoding after the interruption ends.
2. The last buffer that you sent to the codec, before the interruption, may or may not have been successfully written to disk.

Encoding takes place as you repeatedly call the `ExtAudioFileWrite` function with new buffers of audio data. To handle an interruption, respond according to the function's result code, as described here:

- `kExtAudioFileError_CodecUnavailableInputConsumed`—This result code indicates that the last buffer you provided, prior to interruption, was successfully written to disk.

  On receiving this result code, stop calling the `ExtAudioFileWrite` function. If you can resume conversion, wait for an interruption-ended call. In your interruption-end handler, reactivate the session and then resume writing the file.

  Because the last buffer (before interruption) was successfully written to disk, proceed by writing the next buffer.
- `kExtAudioFileError_CodecUnavailableInputNotConsumed`—This result code indicates that the last buffer you provided, prior to interruption, was *not* successfully written to disk.

  Exactly as for the other result code, on receiving this result code, stop calling the `ExtAudioFileWrite` function. If you can resume conversion, wait for an interruption-ended call. In your interruption-end handler, reactivate the session and then resume writing the file.

  Here is where your interruption handling differs from what you do for the other result code: Because the last buffer (before interruption) was *not* successfully written to disk, begin by writing that buffer again.

To check if the AAC codec can resume, obtain the value of the associated converter's `kAudioConverterPropertyCanResumeFromInterruption` property. The value is 1 (can resume) or 0 (cannot resume). You can obtain this value any time after instantiating the converter—immediately after instantiation, upon interruption, or after interruption ends.

If the converter cannot resume, then on interruption you must abandon the conversion. After the interruption ends, or after the user relaunches your application and indicates they want to resume conversion, re-instantiate the extended audio file object and perform the conversion again.

Handling Audio Hardware Route Changes

As your application runs, a user might plug in or unplug a headset, or use a docking station with audio connections. *iPhone Human Interface Guidelines* describes how iOS applications should respond to such events. To implement the recommendations, write audio session code to handle audio hardware route changes.

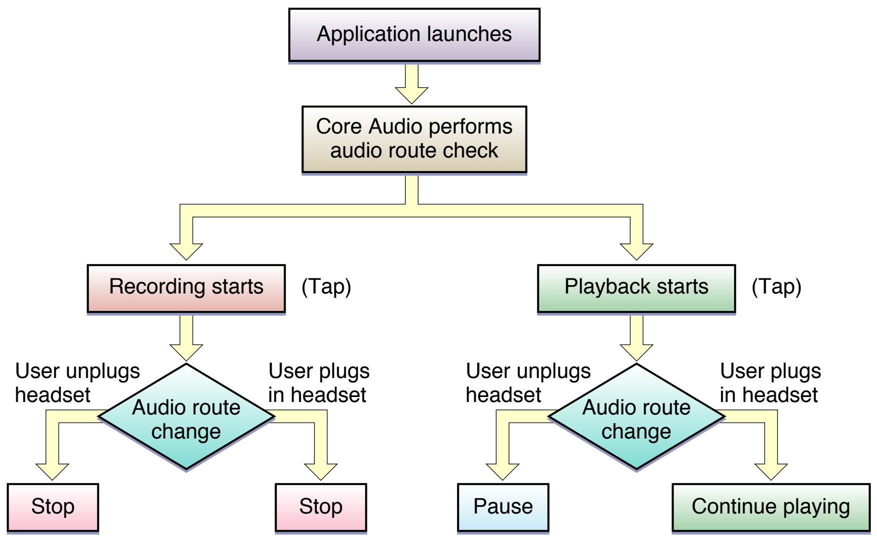
Note: To handle route changes, employ the C-based Audio Session Services interface—even if your other audio session code employs the Objective-C-based AV Foundation framework.

Varieties of Audio Hardware Route Change

An **audio hardware route** is a wired electronic pathway for audio signals. When a user of an iOS device plugs in or unplugs a headset, the system automatically changes the audio hardware route. Your application can listen for such changes by way of the audio session property mechanism.

Figure 4-1 depicts the sequence of events for various route changes during recording and playback. The four possible outcomes, shown across the bottom of the figure, result from actions taken by a property listener callback function that you write.

Figure 4-1 Handling audio hardware route changes



In the figure, The system initially determines the audio route after your application launches. It continues to monitor the active route as your application runs. Consider first the case of a user tapping a Record button in your application, represented by the "Recording starts" box on the left side of the figure.

During recording, the user may plug in or unplug a headset—see the diamond-shaped decision element toward the lower-left of the figure. In response, the system invokes your property listener callback function. To follow Apple's recommendations, your callback would tell your application to stop recording.

The case for playback is similar but has different outcomes, as shown on the right of the figure. If a user unplugs the headset during playback, your callback should pause the audio. If a user plugs in the headset during playback, your callback should simply allow playback to continue.

The *AddMusic* sample code project demonstrates how to implement the playback portion of this behavior.

Responding to Audio Hardware Route Changes

There are three parts to configuring your application to respond to route changes:

1. Implement methods to be invoked upon a route change.
2. Implement a property listener callback function that responds to route changes and uses the methods you implemented in step 1.
3. Register the callback function with the audio session.

For example, say that you write your callback (following Apple guidelines) to pause playback when a user unplugs their headset. After pausing, the callback could display an alert configured with buttons that let the user stop or resume. To support this user interaction, you'd write a method to display the alert and you'd invoke that method from within the callback. You'd register the callback when initializing the audio session during application launch.

When the system invokes a route-change callback, it provides the information you need to figure out which action to take. Base your callback on the `AudioSessionPropertyListener` prototype from audio session services, as shown here:

```
void MyPropertyListener (
   void                    *inClientData,
   AudioSessionPropertyID  inID,
   UInt32                  inDataSize,
   const void              *inData
);
```

For a route change event, the system sends the `kAudioSessionProperty_AudioRouteChange` in the *`inID`* parameter.

The *`inData`* parameter sent to your callback contains a `CFDictionaryRef` object that describes:

- Why the route changed
- What the previous route was

The keys for the dictionary are described in `Audio Route Change Dictionary Keys`. The various reasons why a hardware audio route might have changed—accessed by the `kAudioSession_AudioRouteChangeKey_Reason` key—are listed and described in `Audio Session`

`Route Change Reasons`. The previous route information—accessed by the `kAudioSession_AudioRouteChangeKey_OldRoute` key—is a string value that names the old route, such as "Headphone" or "Speaker".

The information supplied to the callback is usually enough to decide which action to take. For instance, if the reason for the route change is `kAudioSessionRouteChangeReason_OldDeviceUnavailable` and the previous route was `Headphone`, you know that the user just unplugged the headset. If you also need to know what the new route is, call, within your callback, the `AudioSessionGetProperty` function for the `kAudioSessionProperty_AudioRoute` property.

One of the audio hardware route change reasons in iOS is `kAudioSessionRouteChangeReason_CategoryChange`. In other words, a change in audio session category is considered by the system—in this context—to be a route change, and will invoke a route change property listener callback. As a consequence, such a callback—if it is intended to respond only to headset plugging and unplugging—should explicitly ignore this type of route change. The example code in "Defining a Property Listener Callback Function" (page 56) demonstrates this.

Here's a practical example. A well-written recorder/playback application sets the audio session category when beginning playback or recording. Because of this, a route-change property listener callback gets invoked upon starting playback (if you were previously recording) or upon starting recording (if you were previously playing back). Clearly, such an application should not pause or stop each time a user taps Record or Play. To avoid inappropriate pausing or stopping, the callback in this example should branch based on the reason for the route change and simply return if it was a category change.

For a complete example of how to respond to a route change, see "Responding to Audio Hardware Route Changes" (page 55) in "Audio Session Cookbook." To see such a callback in action, download the *AddMusic* sample.

Optimizing for Device Hardware

Using audio session properties, you can optimize your application's audio behavior for device hardware at runtime. This lets your code adapt to the characteristics of the device it's running on, as well as to changes made by the user (such as plugging in a headset or docking the device) as your application runs.

The audio session property mechanism lets you:

- Specify preferred hardware settings for sample rate and I/O buffer duration
- Query many hardware characteristics, among them input and output latency, input and output channel count, hardware sample rate, hardware volume setting, and whether audio input is available
- Write callback functions to respond to property value change events

The most commonly used property value change event is route changes, covered in the previous chapter. You can also write callbacks to listen for changes in hardware output volume and—to support devices that don't have a built-in microphone—changes in the availability of audio input.

Specifying Preferred Audio Hardware Settings

The audio session APIs let you specify preferred hardware sample rate and preferred hardware I/O buffer duration. Table 5-1 describes benefits and costs to consider when specifying such preferences.

Table 5-1 Choosing preferred hardware settings

Setting	Preferred sample rate	Preferred I/O buffer duration
High value	*Example: 44.1 kHz* + High audio quality – Large file size	*Example: 500 mS* + Less-frequent disk access – Longer latency
Low value	*Example: 8 kHz* + Small file size – Low audio quality	*Example: 5 mS* + Low latency – Frequent disk access

For example, as shown in the top-middle cell of the table, you might specify a preference for a high sample rate if audio quality is very important in your application, and if large file size is not a significant issue.

The default audio I/O buffer duration (about 0.02 seconds for 44.1 kHz audio) is appropriate for most applications. It provides enough responsiveness for any sort of user interaction, including fast-action games. Do not set a lower I/O duration unless your application needs it, such as for voice over IP (VOIP).

To set hardware preferences, use the `AudioSessionSetProperty` function along with the `kAudioSessionProperty_PreferredHardwareSampleRate` and `kAudioSessionProperty_PreferredHardwareIOBufferDuration` property identifiers. For a code example, see "Specifying Preferred Hardware I/O Buffer Duration" (page 59).

Important: Although you can safely specify hardware preferences at any time after the audio session is initialized, best practice is to do so while the session is inactive. After you establish hardware preferences, activate the audio session and then query it to obtain the actual values. This final step is important because the system may not be able to provide what you ask for.

Querying Hardware Characteristics

Your application's audio session can tell you about many hardware characteristics of a device. These characteristics can change at runtime. For instance, input sample rate may change when a user plugs in a headset. Callbacks are available for some of these characteristics. For the others, query their values directly.

Important: To obtain meaningful values for hardware characteristics, ensure that the audio session is initialized and active before you issue queries.

Some of the most useful audio session hardware properties are shown in Table 5-2.

Table 5-2 Some useful audio session hardware properties

Audio session property	Description
`kAudioSessionProperty_-CurrentHardwareSampleRate`	The hardware sample rate of the device. Use this property when setting up an audio recording format (see "Obtaining and Using the Hardware Sample Rate" (page 59)). No callback invocation is available for this property.
`kAudioSessionProperty_-CurrentHardwareOutputVolume`	The playback volume of the device.
`kAudioSessionProperty_-CurrentHardwareOutputLatency`	The playback latency of the device. No callback invocation is available for this property.
`kAudioSessionProperty_-AudioInputAvailable`	Whether audio input is available on the device. Use this property to determine if audio recording is possible. For a code example, see "Determining Whether a Device Supports Recording" (page 60).

The full list of audio session properties is described in *Audio Session Services Reference*, in the `Audio Session Services Property Identifiers` enumeration.

Responding to Property Value Change Events

The previous chapter, "Handling Audio Hardware Route Changes" (page 31), went into depth on responding to one particular property value change event—the audio hardware route change. The "Audio Session Cookbook" chapter includes a complete example, "Responding to Audio Hardware Route Changes" (page 55). You can use the same function callback technique to respond to other events, notably:

- Changes made by the user to the hardware playback volume
- Changes to the availability of audio input—meaning the addition or removal of a microphone on devices, such as the iPod touch (2nd generation), that support recording but only when a headset is attached

To respond to playback volume changes, write a property listener callback function that listens for the `kAudioSessionProperty_CurrentHardwareOutputVolume` property. For changes to the availability of audio input, listen for changes to the `kAudioSessionProperty_AudioInputAvailable` property.

Working with Movies and iPod Music

Movie players let you play movies from a file or a network stream. Music players let you play audio content from a user's iPod Library. To use these objects in coordination with your application audio, you must take into account their audio session characteristics.

- Music players (instances of the `MPMusicPlayerController` class) always use a system-supplied audio session.
- Movie players (instances of the `MPMoviePlayerController` class) use your application's audio session by default, but can be configured to use a system-supplied audio session.

Important: In iOS 3.1.3 and earlier, a movie player always uses a system-supplied audio session. To obtain that same behavior in iOS 3.2 and newer, you must set the movie player's `useApplicationAudioSession` property value to `NO`.

Read "Working with Movie Players" (page 40) for more information.

Working with Music Players

To play audio from a user's iPod library along with your own sounds (as described in *iPod Library Access Programming Guide*), you must use a so-called *mixable* category configuration for your audio session. There are two, alternative ways to configure an audio session as mixable:

- Use the `AVAudioSessionCategoryAmbient` (or the equivalent `kAudioSessionCategory_AmbientSound`) category—which is always mixable.
- Use the mixable category override property `kAudioSessionProperty_OverrideCategoryMixWithOthers`, as described in "Fine-Tuning the Category" (page 22), to make an otherwise nonmixable playback category mixable.

Having used one of these options, your sounds will not interrupt a music player—and neither will a music player's sounds interrupt yours.

Important: Do not attempt to use a music player without configuring a mixable category for your audio session.

Because you must use a mixable category configuration, you don't have access to hardware codecs for playback or recording. For details on this, see "How Categories Affect Encoding and Decoding" (page 21).

The system automatically handles route changes and interruptions for music players. You cannot influence this built-in behavior. As long as you correctly manage your application's audio session as described here and in previous chapters, you can rely on a music player to take care of itself, as a user plugs in a headset, an alarm sounds, or a phone call arrives.

You can configure your audio session so that sound from a music player ducks (lowers in volume) when audio from your application plays. For details on ducking and how to enable it, see "Fine-Tuning the Category" (page 22).

For a description of the music player class, see *MPMusicPlayerController Class Reference*.

Working with Movie Players

By default, a movie player shares your application's audio session. This means that, in effect, a movie player transcends the notion of mixing with your application's audio; the movie player's audio behaves as though it belongs to your application. No matter which playback category you choose, and no matter how you configure that category, your audio and the movie player's audio never interrupt each other.

Sharing your audio session also gives you control over how a movie interacts with audio from other applications, such as the iPod. For example, if you set your category to `AVAudioSessionCategoryAmbient` and share your session, iPod audio is not interrupted when a movie starts in your app. Sharing your audio session also lets you specify whether or not movie audio obeys the Ring/Silent switch.

Important: In iOS 3.1.3 and earlier, a movie player always uses a system-supplied audio session. To obtain that same behavior in iOS 3.2 and newer, you must set the movie player's `useApplicationAudioSession` property value to `NO`, as described in the first row of Table 6-1 (page 40).

To configure audio behavior for a movie, determine the behavior you want and then perform appropriate audio session configuration—as described in Table 6-1. For details on setting up your audio session, see "Configuring the Audio Session" (page 19).

Table 6-1 Configuring audio sessions when using a movie player

Desired behavior	Audio session configuration
Playing a movie silences all other audio	■ If your app does not itself play audio, do not configure an audio session. ■ If your app does play audio, configure an audio session and set its mixability according to whether or not you want to mix with iPod and other audio. ■ In either case, tell the movie player to use its own audio session: `myMoviePlayer.useApplicationAudioSession = NO`
Movie and application audio mix, but other audio, including iPod, is silenced	■ Configure an audio session using a *nonmixable* category. ■ Take advantage of the movie player's default `useApplication-AudioSession` value of `YES`
All audio mixes	■ Configure an audio session using a *mixable* category configuration. ■ Take advantage of the movie player's default `useApplication-AudioSession` value of `YES`

Manage your application's audio session as usual in terms of route changes and interruptions, as described in "Handling Audio Hardware Route Changes" (page 31) and "Handling Audio Interruptions" (page 25). Enable ducking, if desired, as described in "Fine-Tuning the Category" (page 22).

If you have configured a movie player to use its own audio session, there's some cleanup to perform. After a movie finishes, or the user dismisses it, do these two steps, in sequence, to regain the ability to play audio:

1. Dispose of the movie player—even if you intend to play the same movie again later.
2. Reactivate your audio session.

For a description of the movie player class, see *MPMoviePlayerController Class Reference*.

Using the Media Player Framework Exclusively

If your application is using a movie player only, or a music player only—and you are not playing your own sounds—then you should not configure an audio session.

If you are using a movie player exclusively, you must tell it to use its own audio session, as follows:

```
myMoviePlayer.useApplicationAudioSession = NO
```

If you are using a movie player *and* a music player, then you probably want to configure how the two interact; for this, you must configure an audio session, even though you are not playing application audio per se. Use the guidance in Table 6-1 (page 40).

Audio Session Cookbook

Each section in this chapter provides example code for implementing a common audio session task. Typically, you would place most of your audio session code in an iPhone view controller class, as is done in the *AddMusic* sample.

To understand how these tasks relate to each other and to your application's life cycle, see the previous chapter, "Configuring the Audio Session" (page 19).

In production code, always check for errors when making API calls. For an example, see Listing 7-4 (page 44). Most examples in this chapter do not illustrate error checking.

Initializing an Audio Session

If you use AV Foundation delegate methods for handling audio interruptions, you can rely on implicit audio session initialization as described in "Initializing the Audio Session" (page 19).

You can instead write a C callback function to handle audio interruptions. To attach that code to the audio session, you must initialize the session explicitly using the `AudioSessionInitialize` function. Do this just once, during application launch.

Note: Your application does not instantiate its audio session. Instead, iOS provides a singleton audio session object to your application, which is available when your application has finished launching.

Listing 7-1 illustrates how to initialize your audio session and register your interruption listener callback function.

Listing 7-1 Initializing a session and registering an interruption callback

```
AudioSessionInitialize (
    NULL,                            // 1
    NULL,                            // 2
    interruptionListenerCallback,    // 3
    userData                         // 4
);
```

Here's how this code works:

1. Use `NULL` to use the default (main) run loop.

2. Use `NULL` to use the default run loop mode.

3. A reference to your interruption listener callback function. See "Responding to Audio Session Interruptions" (page 50) for a description of how to write and use an interruption callback function.

4. Data you intend to be passed to your interruption listener callback function when the audio session object invokes it.

Activating and Deactivating the Audio Session

Activate your audio session as shown in Listing 7-2.

Listing 7-2 Activating an audio session using the AV Foundation framework

```
NSError *activationError = nil;
[[AVAudioSession sharedInstance] setActive: YES error: &activationError];
```

In the rare instance that you want to deactivate your audio session, pass `NO` to the *`setActive`* parameter.

If you prefer to use the C-based Audio Session Services interface, activate your audio session by calling the `AudioSessionSetActive` function. Listing 7-3 shows how.

Listing 7-3 Activating an audio session using Audio Session Services

```
OSStatus activationResult = NULL;
result = AudioSessionSetActive (true);
```

In the rare instance that you want to deactivate your audio session, pass `false` to the `AudioSessionSetActive` function.

Setting the Category

To set the audio session category, call the `setCategory:error:` method as shown in Listing 7-4. For descriptions of all the categories, refer to "Audio Session Categories" (page 63).

Listing 7-4 Setting the audio session category using the AV Foundation framework

```
NSError *setCategoryError = nil;
[[AVAudioSession sharedInstance]
                setCategory: AVAudioSessionCategoryAmbient
                      error: &setCategoryError];

if (setCategoryError) { /* handle the error condition */ }
```

Listing 7-5 shows the equivalent operation using the C-based Audio Session Services interface.

Listing 7-5 Setting the audio session category using Audio Session Services

```
UInt32 sessionCategory = kAudioSessionCategory_AmbientSound;    // 1

AudioSessionSetProperty (
    kAudioSessionProperty_AudioCategory,                        // 2
    sizeof (sessionCategory),                                   // 3
    &sessionCategory                                            // 4
);
```

Here's how this code works:

1. Defines a new variable of type `UInt32` and initializes it with the identifier for the category you want to apply to the audio session.
2. The identifier, or key, for the audio session property you want to set. Other audio session properties are described in `Audio Session Services Property Identifiers`.
3. The size, in bytes, of the property value that you are applying.
4. The category you want to apply to the audio session.

Checking if Other Audio is Playing During App Launch

When a user launches your app, sound may already be playing on the device. This is especially salient if your app is a game. For example, the iPod may be playing music, or Safari may be streaming audio. In such a case, "Using Sound" in *iPhone Human Interface Guidelines* advises you to assume that the user expects the other audio to continue as they play your game.

In such a case, use the Ambient category, which supports mixing with other audio; and do not play your usual background soundtrack.

Listing 7-6 shows how to check whether or not other audio is playing, and to then appropriately set the category for a typical game app. Perform this check after initializing the audio session.

Listing 7-6 Checking if other audio is playing

```
UInt32 otherAudioIsPlaying;                                   // 1
UInt32 propertySize = sizeof (otherAudioIsPlaying);

AudioSessionGetProperty (                                     // 2
    kAudioSessionProperty_OtherAudioIsPlaying,
    &propertySize,
    &otherAudioIsPlaying
);

if (otherAudioIsPlaying) {                                    // 3
    [[AVAudioSession sharedInstance]
                    setCategory: AVAudioSessionCategoryAmbient
                          error: nil];
} else {
    [[AVAudioSession sharedInstance]
                    setCategory: AVAudioSessionCategorySoloAmbient
                          error: nil];
}
```

Here's how this code works:

1. Defines a variable to hold the value of the audio session's `kAudioSessionProperty_OtherAudioIsPlaying` property.

2. Obtains the value of the `kAudioSessionProperty_OtherAudioIsPlaying` property, assigning it to the `otherAudioIsPlaying` variable.

3. If other audio is playing when your application launches, assigns the `AVAudioSessionCategoryAmbient` category to your audio session; otherwise, assigns the `AVAudioSessionCategorySoloAmbient` category.

Modifying Playback Mixing Behavior

Two playback categories that normally silence other audio on the device can be modified to support mixing. These are the `AVAudioSessionCategoryPlayback` (or the equivalent `kAudioSessionCategory_MediaPlayback`) category, and the `AVAudioSessionCategoryPlayAndRecord` (or the equivalent `kAudioSessionCategory_PlayAndRecord`) category. To allow mixing, you set a particular audio session property. Listing 7-7 shows how.

Listing 7-7 Overriding audio mixing behavior

```
OSStatus propertySetError = 0;
UInt32 allowMixing = true;

propertySetError = AudioSessionSetProperty (
                       kAudioSessionProperty_OverrideCategoryMixWithOthers,  //
 1
                       sizeof (allowMixing),                                 //
 2
                       &allowMixing                                          //
 3
                   );
```

Here's how this code works:

1. The identifier, or key, for the audio session property you want to set. Other audio session properties are described in `Audio Session Services Property Identifiers`.

2. The size, in bytes, of the property value that you are applying.

3. The value to apply to the property.

This property has a value of `false` (0) by default. When the audio session category changes, such as during an interruption, the value of this property reverts to `false`. To regain mixing behavior you must then set this property again.

Always check to see if setting this property succeeds or fails, and react appropriately; behavior may change in future releases of iOS.

Ensuring That Audio Continues When the Screen Locks

If you want to ensure that your audio continues when the screen locks and when the Ring/Silent switch is in the "silent" position, you need to assign an appropriate category to your audio session. You may also need to specially configure audio units, if using them for playback, as described in "Configuring Audio Units to Continue Playing When the Screen Locks" (page 47).

You cannot rely on the default audio session, whose category (starting in iOS 2.2) is `AVAudioSessionCategorySoloAmbient` (or equivalently, `kAudioSessionCategory_SoloAmbientSound`).

Use the `AVAudioSessionCategoryPlayback` (or the equivalent `kAudioSessionCategory_MediaPlayback`) category. Assign this category to your audio session as described in "Setting the Category" (page 44). For a complete list of audio session categories and their characteristics, see "Audio Session Categories" (page 63).

Configuring Audio Units to Continue Playing When the Screen Locks

If you are using audio units for playback and want your audio to continue when the screen locks, you may need to perform some additional configuration to support that. This section briefly explains why, and shows how to do it.

- If the system's default audio unit latency of 23 ms (for 44.1 kHz audio) works well for your application, you can take advantage of the lower playback power consumption that is available when the screen locks. To do so, you must perform some configuration on the audio units you are using, as described in "Optimizing for Minimal Power Consumption" (page 47).
- If your application demands shorter audio latency, you instead perform some configuration using the audio session object, as described in "Optimizing for Low Latency" (page 48).

Apple recommends that you optimize for minimal power consumption, unless your application has special needs that demand lower latency, such as VOIP.

Optimizing for Minimal Power Consumption

To optimize for minimal audio playback power consumption, you must adjust the `kAudioUnitProperty_MaximumFramesPerSlice` property value in the audio units you're using, as described here.

By default, the system invokes the I/O unit's render callback with a slice size of 1,024 frames. (See "slice" in *Core Audio Glossary*, and the `kAudioUnitProperty_MaximumFramesPerSlice` property, to learn about slices.) This corresponds to a latency of about 23 ms for 44.1 kHz audio. A latency of 0.02 seconds—while not extremely short—works very well for most audio uses, such as media playback, Internet streaming, and games.

When the screen locks, the system conserves power by increasing the I/O unit's slice size to 4096 frames—equivalent to about 93 ms for 44.1 kHz audio. Specifically, with a larger slice size, the system invokes the I/O unit's render callback less frequently, conserving power. Longer latency while the screen is locked is not a problem, because there is no user interaction. The system manages the slice size for I/O units. Do not set I/O unit slice size yourself.

The system does not, however, adjust the maximum slice size property for other audio units involved in playback. If you are feeding audio to an I/O unit from one or more other audio units, it is up to you to ensure those audio units can handle the larger slice size that is used on screen lock. If you don't do this, audio stops or stutters on screen lock.

For each audio unit involved in playback—other than the I/O unit—set the `kAudioUnitProperty_MaximumFramesPerSlice` property to a value of 4096. The time to do this is when configuring your audio units—before initializing them. If you are using an audio processing graph, do this when configuring the graph—prior to initializing it.

Important: When optimizing for minimal power consumption, do not set a preferred I/O hardware buffer duration—for if you do, playback cannot go into a lower-power mode on screen lock.

Listing 7-8 shows how to support lower-power mode by setting an audio unit's `kAudioUnitProperty_MaximumFramesPerSlice` property, using the `AudioUnitSetProperty` function.

Listing 7-8 Setting an audio unit's maximum-frames-per-slice property

```
UInt32 maximumFramesPerSlice = 4096;

AudioUnitSetProperty (
    mixerUnit,
    kAudioUnitProperty_MaximumFramesPerSlice,
    kAudioUnitScope_Global,
    0,                          // global scope always uses element 0
    &maximumFramesPerSlice,
    sizeof (maximumFramesPerSlice)
);
```

For an example of how to find, instantiate, and configure audio units, see the *iPhoneMultichannelMixerTest* sample.

Optimizing for Low Latency

The system-provided render latency of 23 ms (for 44.1 kHz audio) is too long for certain time-critical applications, such as voice over IP (VOIP). You can reduce latency by setting a preferred I/O hardware buffer duration—see "Optimizing for Device Hardware" (page 35) and "Specifying Preferred Hardware I/O Buffer Duration" (page 59). However, after you set the I/O buffer duration, the system uses that value until you explicitly change it. In other words, audio playback will *not* change to a low-power mode upon screen lock.

Redirecting Output Audio

When using the "play and record" category—which indicates your intention to use both audio input and output—output audio normally goes to the receiver. You can redirect this audio to the speaker at the bottom of the phone in two ways:

1. Using a category routing override. An override reverts to the receiver upon interruption and when a user changes the audio route such as by plugging in or unplugging a headset. If you want to continue using the speaker, you must again invoke the override.

2. Changing the default output route. The new output route remains in effect unless you change the audio session category. This option is available starting in iOS 3.1.

Listing 7-9 shows how to perform a category routing override. Before executing this code, you would set the audio session category to `AVAudioSessionCategoryRecord` (or to the equivalent `kAudioSessionCategory_PlayAndRecord`), as described in "Setting the Category" (page 44).

Listing 7-9 Overriding the output audio route

```
UInt32 audioRouteOverride = kAudioSessionOverrideAudioRoute_Speaker;  // 1

AudioSessionSetProperty (
    kAudioSessionProperty_OverrideAudioRoute,                         // 2
    sizeof (audioRouteOverride),                                      // 3
    &audioRouteOverride                                               // 4
);
```

Here's how this code works:

1. The identifiers you can use for redirecting output audio are described in `Audio Session Category Route Overrides`.

2. The identifier, or key, for the audio session property you want to set.

3. The size, in bytes, of the property value that you are applying.

4. The audio route override that you want to apply to the audio session's category.

If you want to redirect output audio in a more permanent fashion, change the default output as shown in Listing 7-10. Again, you would first set the audio session category to "play and record.

Listing 7-10 Changing the default output audio route

```
UInt32 doChangeDefaultRoute = 1;

AudioSessionSetProperty (
    kAudioSessionProperty_OverrideCategoryDefaultToSpeaker,
    sizeof (doChangeDefaultRoute),
    &doChangeDefaultRoute
);
```

Supporting Bluetooth Audio Input

Starting in iOS 3.1, you can modify the recording audio session categories to allow audio input from a paired Bluetooth device. Listing 7-11 shows how.

Listing 7-11 Modifying a recording category to support Bluetooth input

```
// First, set an audio session category that allows input. Use the
//    AVAudioSessionCategoryRecord or the equivalent kAudioSessionCategory_Record
//    category, or the AVAudioSessionCategoryPlayAndRecord or the equivalent
//    kAudioSessionCategory_PlayAndRecord category. Then proceed as shown here:

UInt32 allowBluetoothInput = 1;

AudioSessionSetProperty (
    kAudioSessionProperty_OverrideCategoryEnableBluetoothInput,
    sizeof (allowBluetoothInput),
    &allowBluetoothInput
);
```

The technique is parallel to the previous code example, "Redirecting Output Audio" (page 49). Before executing the code in Listing 7-11 (page 50), you would set an audio session category that allows input. Use the `AVAudioSessionCategoryRecord` (or the equivalent `kAudioSessionCategory_RecordAudio`) category, or the `AVAudioSessionCategoryPlayAndRecord` (or the equivalent `kAudioSessionCategory_PlayAndRecord`) category.

Responding to Audio Session Interruptions

You can use the AV Foundation framework to respond to interruptions. To do so, implement an "interruption started" and an "interruption ended" delegate method. Alternatively, you can use a C callback function, but doing so is more involved. Using a callback, you:

- Define methods to be invoked upon interruption
- Define an interruption listener callback function
- Register your callback with the audio session

Using the AVAudioSessionDelegate Protocol to Handle Audio Interruptions

This section describes using the `AVAudioSessionDelegate` protocol, suitable for when you are doing audio work with Audio Queue Services, OpenAL, or the I/O audio unit. The next section describes how to instead use the interruption delegate methods provided directly by audio players (see *AVAudioPlayerDelegate Protocol Reference*) and audio recorders (see *AVAudioRecorderDelegate Protocol Reference*).

Listing 7-12 shows simple implementations of the "begin" and "end" interruption methods from the `AVAudioSessionDelegate` protocol, described in *AVAudioSessionDelegate Protocol Reference*. When your "begin" delegate gets called, your audio has already been stopped and your audio session has already been deactivated.

Listing 7-12 Implementing AVAudioSessionDelegate interruption methods

```
- (void) beginInterruption {
    if (playing) {
        playing = NO;
        interruptedWhilePlaying = YES;
        [self updateUserInterface];
    }
}

NSError *activationError = nil;
- (void) endInterruption {
    if (interruptedWhilePlaying) {
       [[AVAudioSession sharedInstance] setActive: YES error: &activationError];
        [player play];
        playing = YES;
        interruptedWhilePlaying = NO;
        [self updateUserInterface];
    }
}
```

The `beginInterruption` method updates application state and user interface.

The `endInterruption` method reactivates the audio session, resumes playback, and updates application state and user interface.

Handling Interruptions with the AVAudioPlayer Class

The `AVAudioPlayer` class, first available in iOS 2.2, provides its own interruption delegate methods, described in *AVAudioPlayerDelegate Protocol Reference*. Likewise, the `AVAudioRecorder` class, first available in iOS 3.0, provides its own interruption delegate methods, described *AVAudioRecorderDelegate Protocol Reference*. This section describes how to handle interruptions with the `AVAudioPlayer` class. You use a very similar approach for the `AVAudioRecorder` class.

When your interruption-started delegate gets called, your audio player has already been paused and your audio session has already been deactivated. You can provide an implementation such as that shown in Listing 7-13.

Listing 7-13 An interruption-started delegate method for an audio player

```
- (void) audioPlayerBeginInterruption: (AVAudioPlayer *) player {
    if (playing) {
        playing = NO;
        interruptedOnPlayback = YES;
        [self updateUserInterface];
    }
}
```

After checking to ensure that the audio player was indeed playing when the interruption arrived, this method updates your application's state and then updates the user interface. (The system automatically pauses the audio player.)

If a user ignores a phone call, the system automatically reactivates your audio session and your interruption-ended delegate method gets invoked. Listing 7-14 shows a simple implementation of this method.

Listing 7-14 An interruption-ended delegate method for an audio player

```
- (void) audioPlayerEndInterruption: (AVAudioPlayer *) player {
    if (interruptedOnPlayback) {
        [player prepareToPlay];
        [player play];
        playing = YES;
        interruptedOnPlayback = NO;
    }
}
```

Defining Interruption Methods

Note: The remainder of this section on responding to audio interruptions applies if you are handling playback or recording interruptions with a C-language interruption listener callback function.

To respond effectively to audio interruptions, your application needs methods that take appropriate action upon interruption-begin and interruption-end. Some of these actions—such as to stop recording—are ones you define anyway for your application. Others—such as to save playback state—may be ones that you invoke only from the interruption listener callback function.

Methods to be invoked only by a callback do not have associated user interface. Otherwise, they are like other control methods. For examples of audio control methods with and without a user interface, see the `recordOrStop:`, `playOrStop:`, `pausePlayback:`, and `resumePlayback:` methods in the `AudioViewController.m` class file in the *SpeakHere* sample code project. The `resumePlayback:` method, which is not directly invoked by a user interface element, is reproduced in Listing 7-15.

Listing 7-15 An interruption method

```
- (void) resumePlayback {

    UInt32 sessionCategory = kAudioSessionCategory_MediaPlayback;  // 1
    AudioSessionSetProperty (                                      // 2
        kAudioSessionProperty_AudioCategory,                       // 3
        sizeof (sessionCategory),                                  // 4
        &sessionCategory                                           // 5
    );
    AudioSessionSetActive (true);                                  // 6

    [self.audioPlayer resume];                                     // 7
}
```

Here's how this code works:

1. Defines a variable to represent the audio session category and initializes it to the “media playback” category identifier. This category indicates the intent that other audio in the phone should be silenced when your audio session activates. You set the category prior to activating the audio session, which you do in line 6.
2. The `AudioSessionSetProperty` function assigns a value to a designated property in your audio session object.
3. The audio session property you want to set a value for—here, the “audio category” property.
4. The size of the value you are assigning to the property.
5. The address of the variable containing the category identifier.
6. Activates the audio session, which you do just before resuming playback.

> **Note:** Activation is required here, because your audio session was deactivated when the interruption started. If you do not activate your audio session after an interruption ends, audio features, including playback and recording, do not work in your application.

7. Invokes the `resume` method of the `audioPlayer` object.

Defining an Interruption Listener Callback Function

You define your interruption listener callback function to respond to the two possible interruption states, identified by the audio session constants `kAudioSessionBeginInterruption` and `kAudioSessionEndInterruption`. Listing 7-16 illustrates a basic implementation.

Listing 7-16 An interruption listener callback function

```
void interruptionListenerCallback (
    void    *inUserData,                                        // 1
    UInt32  interruptionState                                   // 2
) {
    AudioViewController *controller =
        (AudioViewController *) inUserData;                     // 3

    if (interruptionState == kAudioSessionBeginInterruption) {          // 4

        if (controller.audioRecorder) {
            [controller recordOrStop: (id) controller];         // 5
        } else if (controller.audioPlayer) {
            [controller pausePlayback];                         // 6
            controller.interruptedOnPlayback = YES;             // 7
        }

    } else if ((interruptionState == kAudioSessionEndInterruption) &&
                                    controller.interruptedOnPlayback) { // 8
        [controller resumePlayback];
        controller.interruptedOnPlayback = NO;
    }
}
```

Here's how this code works:

1. A pointer to data that you provide when initializing your audio session.

   In an Objective-C class file, such as a view controller class, you place the interruption listener callback function outside of the class implementation block. Because of this, the callback needs a reference to the controller object to be able to send messages to it. You provide this reference when initializing your audio session, as described in Listing 7-1 (page 43).

2. The interruption state, which is a constant from the `Audio Session Interruption States` enumeration.

3. Initializes an `AudioViewController` object instance to the reference passed in by the *`inUserData`* parameter.

4. If true, an audio session interruption has just begun. Your recording or playback has been stopped.

5. If currently recording, flushes recording buffers and updates the user interface to show the stopped condition.

6. If currently playing, saves playback state and updates the user interface to show the stopped (or, effectively, paused) condition.

7. Sets a flag to indicate that playback was interrupted. This flag gets used if the callback is invoked again with an "end interruption" state.

8. If the interruption was removed, and the app had been playing, resumes playback.

Registering Your Interruption Callback with the Audio Session

Your audio session needs to be told about your interruption listener callback function so that it can send it messages. You register your callback during session initialization, as described in "Initializing an Audio Session" (page 43).

Responding to Interruptions When Using OpenAL

When using OpenAL for audio playback, you must manage the OpenAL context , as shown in Listing 7-17. Register this callback function with the audio session as described in "Initializing an Audio Session" (page 43).

Listing 7-17 Managing the OpenAL context during an audio interruption

```
void openALInterruptionListener (
    void   *inClientData,
    UInt32 inInterruptionState
) {
    if (inInterruptionState == kAudioSessionBeginInterruption) {
        alcMakeContextCurrent (NULL);
     } else if (inInterruptionState == kAudioSessionEndInterruption) {
        alcMakeContextCurrent (myContext);
     }
    // other interruption-listener handling code
}
```

To make your code compatible with older versions of iOS, conditionalize it as shown in Listing 7-18.

Listing 7-18 Making OpenAL interruption code compatible with older versions of iOS

```
- (NSInteger) getMajorOSVersion {
    NSString *versionString = [[UIDevice currentDevice] systemVersion];
    return [[[versionString componentsSeparateByString:@"."] objectAtIndex:0]
intValue];
}

void platformIndependentOpenALInterruptionListener (
    void   *inClientData,
    UInt32 inInterruptionState
) {
    if ([self getMajorOSVersion] >= 3) {
        // 3.0 OS mechanism -- use alcMakeContextCurrent
    } else {
        if (inInterruptionState == kAudioSessionBeginInterruption) {
            // Save state of OpenAL context and related sources
            StoreMyContextState (myContext);
            alcDestroyContext (myContext);
        } if (inInterruptionState == kAudioSessionEndInterruption) {
            ALCcontext *myContext = alcCreateContext (myOpenALDevice, NULL);
            // Restore OALContext State and related OALSources
            RestoreMyContextState (myContext);
        }
    }
    // other interruption-listener handling code
}
```

Responding to Audio Hardware Route Changes

To respond to route changes, you employ a property listener callback function. The system invokes the callback when a user plugs in or unplugs a headset, or docks or undocks the device—thereby adding or removing an audio connection. The system also invokes the property listener callback when a Bluetooth device connects or disconnects.

The example code in this section includes configuring and presenting an alert that lets a user choose how to proceed when audio playback is paused following a route change.

Defining Route Change Methods

To present an alert that provides a resume-or-stop choice, implement a `UIAlertView` delegate method as shown in Listing 7-19. You can employ this method in the property listener callback function as shown in "Defining a Property Listener Callback Function" (page 56). (Refer to *UIAlertViewDelegate Protocol Reference* for the `alertView:clickedButtonAtIndex:` method's description.) Code similar to this is used in the *AddMusic* sample.

Listing 7-19 A UIAlertView delegate method

```
- (void) alertView: routeChangeAlertView
             clickedButtonAtIndex: buttonIndex {        // 1

    if ((NSInteger) buttonIndex == 1) {
        [self resumePlayback];                          // 2
    } else {
        [self playOrStop: self];                        // 3
    }

    [routeChangeAlertView release];                     // 4
}
```

Here's how this code works:

1. The *buttonIndex* parameter contains the zero-indexed integer corresponding to the button the user tapped. The left-most button in the alert has index 0.
2. Invokes the `resumePlayback` method in the current object when the user taps the rightmost button.
3. Responds to the user tapping the leftmost of the two buttons in the alert. Invokes the `playOrStop:` method in the current object.
4. Releases the `UIAlertView` object, which was allocated earlier in the property listener callback function (see "Defining a Property Listener Callback Function").

Defining a Property Listener Callback Function

To respond to a route change, a property listener callback function must:

- Identify the nature of the route change
- Branch, depending on the specific route change and the current audio context (for example, recording, playback, or stopped)
- Take or invoke appropriate action

Listing 7-20 shows one way to write a callback that responds to route changes. To see a similar callback function in context, refer to the *AddMusic* sample code project. With appropriate modifications, you can use a callback like this to respond to the other audio session state changes, which include changes to hardware audio output volume and whether audio input is available.

Listing 7-20 A property listener callback function for route changes

```
void audioRouteChangeListenerCallback (
    void                   *inUserData,                          // 1
    AudioSessionPropertyID inPropertyID,                         // 2
    UInt32                 inPropertyValueSize,                  // 3
    const void             *inPropertyValue                      // 4
) {
    if (inPropertyID != kAudioSessionProperty_AudioRouteChange) return; // 5

    MainViewController *controller = (MainViewController *) inUserData; // 6
    if (controller.appSoundPlayer.playing == 0 ) {                      // 7
        return;
    } else {
        CFDictionaryRef routeChangeDictionary = inPropertyValue;        // 8
        CFNumberRef routeChangeReasonRef =
                CFDictionaryGetValue (
                    routeChangeDictionary,
                    CFSTR (kAudioSession_AudioRouteChangeKey_Reason)
                );

        SInt32 routeChangeReason;
        CFNumberGetValue (
            routeChangeReasonRef, kCFNumberSInt32Type, &routeChangeReason
        );

        if (routeChangeReason ==
                kAudioSessionRouteChangeReason_OldDeviceUnavailable) {  // 9

            [controller.appSoundPlayer pause];

            UIAlertView *routeChangeAlertView =
                [[UIAlertView alloc]
                    initWithTitle: @"Playback Paused"
                          message: @"Audio output was changed."
                         delegate: controller
                cancelButtonTitle: @"Stop"
                otherButtonTitles: @"Play", nil];
            [routeChangeAlertView show];
        }
    }
}
```

Here's how this code works:

1. A pointer to data that you provide when initializing your audio session.

   In an Objective-C class file, such as a view controller class, you place the property listener callback function outside of the class implementation block. Because of this, the callback needs a reference to the controller object to be able to send messages to it. You provide this reference when initializing your audio session, as described in Listing 7-21 (page 58).

2. The identifier for the property that this callback function gets notified about.

3. The size, in bytes, of the data in the *inPropertyValue* parameter.

4. The current value of the property that this callback function is monitoring. Because the property being monitored is the `kAudioSessionProperty_AudioRouteChange` property, this value is a `CFDictionary` object.

5. Ensures that the callback was invoked for the correct audio session property change.

6. Initializes a `MainViewController` object instance to the reference passed in by the *inUserData* parameter. This allows the callback to send messages to your view controller object—typically defined in the same file that implements the callback.

7. If application sound is not playing, there's nothing to do, so return.

8. This line and the next several lines determine the reason for the route change. The one route change of interest in this playback-only example is that an output device, such as a headset, was removed.

9. If an output device was indeed removed, then pause playback and display an alert that allows the user to stop or resume playback. Release of the `UIAlertView` object takes place in the method shown in Listing 7-19 (page 56).

Registering Your Property Listener Callback with the Audio Session

Your application can listen for hardware and route change events by way of the property mechanism in Audio Session Services. For example, to listen for route change events, you register a callback function with your audio session object, as shown in Listing 7-21.

Listing 7-21 Registering a property listener callback in Audio Session Services

```
AudioSessionPropertyID routeChangeID =
                        kAudioSessionProperty_AudioRouteChange;    // 1
AudioSessionAddPropertyListener (                                  // 2
    routeChangeID,                                                 // 3
    propertyListenerCallback,                                      // 4
    userData                                                       // 5
);
```

Here's how this code works:

1. Declares and initializes a variable to the identifier for the property you want to monitor.

2. Registers your property listener callback function with your initialized audio session.

3. The identifier for the property you want to monitor.

4. A reference to your hardware listener callback function.

5. Data you want the audio session to pass back to your callback function.

Querying and Using Audio Hardware Characteristics

Hardware characteristics of an iPhone can change while your application is running, and can differ from device to device. When using the built-in microphone for an original iPhone, for example, recording sample rate is limited to 8 kHz; attaching a headset and using the headset microphone provides a higher sample rate. Newer iPhone models support higher hardware sample rates for the built-in microphone.

Before you specify preferred hardware characteristics, ensure that the audio session is inactive. After establishing your preferences, activate the session and then query it to determine the actual characteristics. This final step is important because in some cases, the system is unable to provide what you ask for.

Specifying Preferred Hardware I/O Buffer Duration

You can specify preferred hardware sample rate and preferred hardware I/O buffer duration using the `AVAudioSession` class or by using Audio Session Services. Listing 7-22 shows how to set a preferred I/O buffer duration using the `AVAudioSession` class. To set preferred sample rate you'd use similar code.

Listing 7-22 Specifying preferred I/O buffer duration using the AVAudioSession class

```
NSError *setPreferenceError = nil;
NSTimeInterval preferredBufferDuration = 0.005;
[[AVAudiosession sharedInstance]
            setPreferredIOBufferDuration: preferredBufferDuration
                                   error: &setPreferenceError];
```

Listing 7-23 shows how to accomplish the same thing using the C-based Audio Session Services. To set preferred sample rate you'd use similar code.

Listing 7-23 Specifying a preferred I/O buffer duration using Audio Session Services

```
Float32 preferredBufferDuration = 0.005;                      // 1
AudioSessionSetProperty (                                     // 2
    kAudioSessionProperty_PreferredHardwareIOBufferDuration,
    sizeof (preferredBufferDuration),
    &preferredBufferDuration
);
```

Here's how this code works:

1. Declares and initializes a value for I/O buffer duration, in seconds.

2. Sets the preferred I/O buffer duration.

After establishing a hardware preference, always ask the hardware for the actual value, as shown in Listing 7-25 (page 60); the system may not be able to provide what you ask for.

Obtaining and Using the Hardware Sample Rate

As part of setting up for recording audio, you need to obtain the current audio hardware sample rate and apply it to your audio data format. Listing 7-24 shows how. You would typically place this code in the implementation file for a recording class. Use similar code to obtain other hardware properties including the input and output number of channels.

Before you query the audio session for current hardware characteristics, ensure that the session is active.

Listing 7-24 Obtaining the current audio hardware sample rate using the AVAudioSession class

```
double sampleRate;
sampleRate = [[AVAudiosession sharedInstance] currentHardwareSampleRate];
```

You can accomplish the same thing using the C-based Audio Session Services, as shown in Listing 7-25.

Listing 7-25 Obtaining the current audio hardware sample rate using Audio Session Services

```
UInt32 propertySize = sizeof (self.hardwareSampleRate);    // 1

AudioSessionGetProperty (                                  // 2
    kAudioSessionProperty_CurrentHardwareSampleRate,
    &propertySize,
    &hardwareSampleRate
);
```

Here's how this code works:

1. Declares and initializes a `UInt32` variable to the size, in bytes, of the `hardwareSampleRate` instance variable—the instance variable belongs to the object that defines this method. This instance variable is declared (in a header file) as a `Float64` value. For a complete example, refer to the *SpeakHere* sample code project.
2. Obtains the current audio hardware sample rate by querying the audio session object using the `kAudioSessionProperty_CurrentHardwareSampleRate` property identifier. The sample rate is stored in the `hardwareSampleRate` instance variable.

Determining Whether a Device Supports Recording

If your application is running on an iPhone, recording is supported. If running on an iPod touch (2nd generation), however, recording is possible only when appropriate accessory hardware is attached. Listing 7-27 demonstrates how to determine if recording is possible on a device.

Test for audio input support after you initialize the audio session—which you typically do upon application launch. You should also implement the `inputIsAvailableChanged:` method from the `AVAudioSessionDelegate` protocol, described in *AVAudioSessionDelegate Protocol Reference*, to get messages about changes in input availability.

Listing 7-26 Using the AVAudioSession class to determine if a device supports audio recording

```
BOOL inputAvailable;
inputAvailable = [[AVAudioSession sharedInstance] inputIsAvailable];
```

You can accomplish the same thing using the C-based Audio Session Services, as shown in Listing 7-27.

Listing 7-27 Using Audio Session Services to determine if a device supports audio recording

```
UInt32 audioInputIsAvailable;                            // 1
UInt32 propertySize = sizeof (audioInputIsAvailable);    // 2

AudioSessionGetProperty (                                // 3
    kAudioSessionProperty_AudioInputAvailable,
    &propertySize,
    &audioInputIsAvailable
);
```

Here's how this code works:

1. Declares a variable to hold the value of the `kAudioSessionProperty_AudioInputAvailable` property.

2. Declares a variable and initialize it to the property size.

3. Gets the value of the `kAudioSessionProperty_AudioInputAvailable` property from your initialized audio session. The `audioInputIsAvailable` variable is set to a nonzero value if audio input is available.

Running Your App in the Simulator

Because of the characteristics of the Simulator (as described in "Developing with the Audio Session APIs" (page 18)), you may want to conditionalize your code to allow partial testing in the Simulator.

One approach is to branch based on the return value of an API call. For example, calling the `AudioSessionGetProperty` function with the `kAudioSessionProperty_AudioRoute` property identifier, to obtain the current audio route, fails in the Simulator. Ensure that you are checking and appropriately responding to the result codes from all of your audio session function calls; the result codes may indicate why your application works correctly on a device but fails in the Simulator.

In addition to correctly using audio session result codes, you can employ preprocessor conditional statements to hide certain code when running in the Simulator. Listing 7-28 shows how to do this.

Listing 7-28 Using preprocessor conditional statements

```
#if TARGET_IPHONE_SIMULATOR
#warning *** Simulator mode: audio session code works only on a device
    // Execute subset of code that works in the Simulator
#else
    // Execute device-only code as well as the other code
#endif
```

Providing Usage Guidelines

A user may not want an application to be interrupted by a competing audio session—for instance, when running an audio recorder to capture a presentation.

There is no programmatic way to ensure that an audio session is never interrupted. The reason is that iOS always gives priority to the phone. iOS also gives high priority to certain alarms and alerts—you wouldn't want to miss your flight now, would you?

The solution to guaranteeing an uninterrupted recording is for users to deliberately silence their iPhones by taking the following steps:

1. In the Settings application, ensure that the iPhone has Airplane Mode turned on.

2. In the Calendar application, ensure that there are no event alarms enabled during the planned recording period.

3. In the Clock application, ensure that no clock alarms are enabled during the planned recording period.

4. Do not move the Ring/Silent switch during the recording. When changing Ring/Silent mode, an iPhone may vibrate, depending on user settings.

5. Do not plug in or unplug a headset during recording. Likewise, do not dock or undock the device during recording.
6. Do not plug the iPhone into a power source during the recording. When an iPhone gets plugged into power, it beeps or vibrates, according to user settings.

A user guide is a good place to communicate these steps to your users.

Audio Session Categories

You tell iOS your application's audio intentions by designating a category for your audio session. Table A-1 provides details on each of the categories. The default category, `kAudioSessionCategory_SoloAmbientSound`, is shaded. For an explanation of how categories work, see "Choosing the Best Category" (page 20). For an explanation of the mixing override switch, see "Fine-Tuning the Category" (page 22).

Table A-1 Audio session category behavior

Category identifiers*	Silenced by the Ring/Silent switch and by screen locking	Allows audio from other applications	Allows audio input (recording) and output (playback)
`AVAudioSessionCategoryAmbient` `kAudioSessionCategory_AmbientSound`	Yes	Yes	Output only
`AVAudioSessionCategorySoloAmbient` `kAudioSessionCategory_SoloAmbientSound`	Yes	No	Output only
`AVAudioSessionCategoryPlayback` `kAudioSessionCategory_MediaPlayback`	No	No by default; yes by using override switch	Output only
`AVAudioSessionCategoryRecord` `kAudioSessionCategory_RecordAudio`	No (recording continues with the screen locked)	No	Input only
`AVAudioSessionCategoryPlayAndRecord` `kAudioSessionCategory_PlayAndRecord`	No	No by default; yes by using override switch	Input and output
`AVAudioSessionCategoryAudioProcessing` `kAudioSessionCategory_AudioProcessing`	–	No	No input and no output

*In each row, the first identifier is for the Objective-C-based `AVAudioSession` interface; the second identifier is for the C-based Audio Session Services interface.

Notes: Versions of iOS earlier than iOS 2.2 did not include the `kAudioSessionCategory_SoloAmbientSound` category and instead used the `kAudioSessionCategory_MediaPlayback` category as the default. Two deprecated categories are not shown in the table: The deprecated `kAudioSessionCategory_UserInterfaceSoundEffects` category is equivalent to `AVAudioSessionCategoryAmbient`; the deprecated `kAudioSessionCategory_LiveAudio` category is equivalent to `AVAudioSessionCategoryPlayback`.

Document Revision History

This table describes the changes to *Audio Session Programming Guide*.

Date	Notes
2010-07-09	Minor changes.
2010-04-12	Updated for iOS 3.2 by describing changes in the behavior of movie player audio sessions. See "Working with Movies and iPod Music" (page 39).
2010-01-29	Minor changes to description for "Optimizing for Minimal Power Consumption" (page 47) and "Specifying Preferred Audio Hardware Settings" (page 35).
2010-01-20	Improved "About Configuring Audio Behavior" (page 9).
	Added code recipe: "Checking if Other Audio is Playing During App Launch" (page 45).
	Added code recipe: "Configuring Audio Units to Continue Playing When the Screen Locks" (page 47).
	Added best practice information for working with audio hardware, as follows: added "Important" note to "Specifying Preferred Audio Hardware Settings" (page 35); added "Important" note to "Querying Hardware Characteristics" (page 36); improved the code recipes in "Querying and Using Audio Hardware Characteristics" (page 58).
2009-10-19	Provided additional information on Simulator behavior in regard to audio sessions in "Developing with the Audio Session APIs" (page 18).
	Added some tips for working with the characteristics of the Simulator in "Running Your App in the Simulator" (page 61).
2009-09-09	Updated for iOS 3.1 to include descriptions of new audio session properties.
	Expanded document by adding four new chapters: "Handling Audio Interruptions" (page 25), "Handling Audio Hardware Route Changes" (page 31), "Optimizing for Device Hardware" (page 35), and "Working with Movies and iPod Music" (page 39).
	Added new material to "Configuring the Audio Session" (page 19) including "How Categories Affect Encoding and Decoding" (page 21) and "Fine-Tuning the Category" (page 22).
	Added new sections in "Audio Session Basics" (page 13): "What Is an Audio Session Category?" (page 14) and "The Two Audio Session APIs" (page 17).

Date	Notes
	Expanded "Audio Session Cookbook" (page 43) to include examples to support new material.
2008-11-13	New document for iOS 2.2 that explains how to use an iOS application's audio session.