
iOS Application Programming Guide

General



2010-08-20



Apple Inc.
© 2010 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

App Store is a service mark of Apple Inc.

Apple, the Apple logo, Bonjour, Cocoa, Finder, Instruments, iPhone, iPod, iPod touch, iTunes, Keychain, Mac, Mac OS, Objective-C, Quartz, Safari, Sand, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

iPad is a trademark of Apple Inc.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

OpenGL is a registered trademark of Silicon Graphics, Inc.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Simultaneously published in the United States and Canada.

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

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

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

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

Contents

Introduction About iOS Application Design 9

Starting the Design Process for Your Application 9
Supporting Common Application Behaviors 10
Meeting the App Store and System Requirements 10
Tuning Performance for the Underlying Device 11
See Also 11

Chapter 1 The Application Runtime Environment 13

Fast Launch, Short Use 13
Specialized System Behaviors 13
 The Virtual Memory System 14
 The Automatic Sleep Timer 14
 Multitasking Support 14
Security 15
 The Application Sandbox 15
 File Protection 15
 Keychain Data 16
The File System 16
 A Few Important Application Directories 16
 A Case-Sensitive File System 18
 Sharing Files with the User's Desktop Computer 18
Backup and Restore 18
 What Is Backed Up? 18
 Files Saved During Application Updates 19
The Simulator 19
Determining the Available Hardware Support 20

Chapter 2 The Core Application Design 23

Fundamental Design Patterns 23
The Core Application Objects 24
The Application Life Cycle 27
 The Main Function 28
 The Application Delegate 29
 Understanding an Application's States and Transitions 29
Multitasking 36
 Checklist for Supporting Multitasking 36
 Responding to System Changes While in the Background 36
 Opting Out of Background Execution 38
Windows, Views, and View Controllers 39

The Event-Handling System	40
The Graphics and Drawing System	42
The Text System	42
Audio and Video Support	44
Integration with the Hardware and System Applications	44

Chapter 3 **Implementing Standard Application Behaviors 47**

Observing Low-Memory Warnings	47
Preserving the Application's Current State	48
Launching in Landscape Mode	48
Files and the File System	49
Getting Paths to Application Directories	49
Sharing Files with the User	51
Working with Protected Files	51
Opening Files Whose Type is Unknown	52
Implementing Support for Custom File Formats	54
Communicating with Other Applications	56
Implementing Custom URL Schemes	57
Registering Custom URL Schemes	57
Handling URL Requests	58
Displaying Application Preferences	60
Turning Off Screen Locking	61

Chapter 4 **Executing Code in the Background 63**

Preparing Your Application to Execute in the Background	63
Determining Whether Multitasking Support is Available	63
Declaring the Background Tasks You Support	64
Supporting Background State Transitions	64
Being a Responsible, Multitasking-Aware Application	65
Initiating Background Tasks	66
Completing a Long-Running Task in the Background	66
Scheduling the Delivery of Local Notifications	68
Receiving Location Events in the Background	69
Playing Background Audio	70
Implementing a VoIP Application	70

Chapter 5 **Supporting High-Resolution Screens 73**

Checklist for Supporting High-Resolution Screens	73
Points Versus Pixels	73
Drawing Improvements That You Get for Free	74
Updating Your Image Resource Files	75
Loading Images into Your Application	75
Using an Image View to Display Images	76

- Updating Your Application's Icons and Launch Images 76
- Updating Your Custom Drawing Code 76
 - Creating High-Resolution Bitmap Images Programmatically 76
 - Tweaking Native Content for High-Resolution Screens 77
 - Accounting for Scale Factors in Core Animation Layers 77
 - Drawing High-Resolution Content Using OpenGL ES 78

Chapter 6 **Implementing Application Preferences 81**

- Guidelines for Preferences 81
- The Preferences Interface 82
- The Settings Bundle 83
 - The Settings Page File Format 84
 - Hierarchical Preferences 85
 - Localized Resources 86
- Adding and Modifying the Settings Bundle 86
 - Adding the Settings Bundle 86
 - Preparing the Settings Page for Editing 87
 - Configuring a Settings Page: A Tutorial 87
 - Creating Additional Settings Page Files 91
- Accessing Your Preferences 91
- Debugging Preferences for Simulated Applications 92

Chapter 7 **Build-Time Configuration Details 93**

- The Application Bundle 93
- The Information Property List 95
- iTunes Requirements 97
 - Declaring the Required Device Capabilities 98
 - Application Icons 100
- Application Launch Images 101
 - Providing Launch Images for Different Orientations 102
 - Providing Device-Specific Launch Images 103
 - Providing Launch Images for Custom URL Schemes 104
- Internationalizing Your Application 104

Chapter 8 **Tuning for Performance and Responsiveness 107**

- Do Not Block the Main Thread 107
- Use Memory Efficiently 107
 - Reduce Your Application's Memory Footprint 108
 - Allocate Memory Wisely 108
- Floating-Point Math Considerations 109
- Reduce Power Consumption 110
- Tune Your Code 111
- Improve File Access Times 111

Tune Your Networking Code	112
Tips for Efficient Networking	112
Using Wi-Fi	113
The Airplane Mode Alert	113

Document Revision History 115

Figures, Tables, and Listings

Chapter 1 **The Application Runtime Environment 13**

Table 1-1	Directories of an iOS application	17
Table 1-2	Identifying available hardware features	20

Chapter 2 **The Core Application Design 23**

Figure 2-1	Key objects in an iOS application	25
Figure 2-2	Application life cycle	27
Figure 2-3	Launching into the active state	30
Figure 2-4	Moving from the foreground to the background	32
Figure 2-5	Handling application interruptions	33
Figure 2-6	Transitioning from the background to the foreground	35
Figure 2-7	Processing events in the main run loop	41
Figure 2-8	Several different keyboards and input methods	43
Table 2-1	Design patterns used by iOS applications	23
Table 2-2	The role of objects in an iOS application	25
Table 2-3	Application states	29
Table 2-4	Notifications delivered to waking applications	37
Table 2-5	System integration technologies	44
Listing 2-1	The <code>main</code> function of an iOS application	28

Chapter 3 **Implementing Standard Application Behaviors 47**

Figure 3-1	Defining a custom URL scheme in the <code>Info.plist</code> file	58
Table 3-1	Commonly used search path constants	50
Table 3-2	Keys and values of the <code>CFBundleURLTypes</code> property	57
Listing 3-1	Getting the path to the application's <code>Documents</code> directory	50
Listing 3-2	Document type information for a custom file format	55
Listing 3-3	Handling a URL request based on a custom scheme	59

Chapter 4 **Executing Code in the Background 63**

Table 4-1	Configuring stream interfaces for VoIP usage	71
Listing 4-1	Checking for background support on earlier versions of iOS	63
Listing 4-2	Starting a background task at quit time	67
Listing 4-3	Scheduling an alarm notification	68

Chapter 5 **Supporting High-Resolution Screens 73**

Listing 5-1	Initializing a renderbuffer's storage and retrieving its actual dimensions	78
-------------	--	----

Chapter 6 **Implementing Application Preferences 81**

Figure 6-1	Organizing preferences using child panes 85
Figure 6-2	Formatted contents of the <code>Root.plist</code> file 87
Figure 6-3	A root Settings page 88
Table 6-1	Preference element types 82
Table 6-2	Contents of the <code>Settings.bundle</code> directory 83
Table 6-3	Root-level keys of a preferences Settings Page file 84
Listing 6-1	Accessing preference values in an application 92

Chapter 7 **Build-Time Configuration Details 93**

Figure 7-1	The Properties pane of a target's Info window 96
Figure 7-2	The information property list editor 97
Figure 7-3	The Language preference view 105
Table 7-1	A typical application bundle 93
Table 7-2	Dictionary keys for the <code>UIRequiredDeviceCapabilities</code> key 98
Table 7-3	Sizes for images in the <code>CFBundleIconFiles</code> key 100
Table 7-4	Typical launch image dimensions 102
Table 7-5	Launch image orientation modifiers 102
Listing 7-1	The contents of a language-localized subdirectory 106

Chapter 8 **Tuning for Performance and Responsiveness 107**

Table 8-1	Tips for reducing your application's memory footprint 108
Table 8-2	Tips for allocating memory 109

About iOS Application Design

This document provides fundamental information about how to design and implement iOS applications. You should use this document as a starting point for learning about iOS and the steps involved in creating applications for it. Specifically, this document describes the infrastructure present in all applications and offers guidance on how to structure your application to run well on the platform.



Almost all of the information in this document applies to applications running on any type of iOS device. Where differences do exist, this document calls out the differences and explains what options exist for each type of device. For additional information about creating iPad applications, see *iPad Programming Guide*.

Note: This document was previously titled *iPhone Application Programming Guide*.

Starting the Design Process for Your Application

Designing an iOS application requires at least a basic understanding of a few key principles:

- The UIKit framework provides the core infrastructure for managing and running your application. Application customization comes primarily through interactions with the classes of this framework.
- The system frameworks use well-defined design patterns. Understanding those design patterns (which are documented in *Cocoa Fundamentals Guide*) is essential for interacting with the system frameworks and is often useful for implementing your own code, too.

- Most other frameworks provide services or additional features that you can incorporate as needed. Some frameworks (such as Foundation and Core Graphics) are included in all Xcode projects, but most others must be added to your project explicitly. For a list of available technologies and the documents that show you how to use them, see *iOS Technology Overview*.

The focus of this document is on the core infrastructure provided by UIKit and how it runs your application. Understanding this infrastructure is crucial to creating applications that work well within the system. This document also addresses the things you must do outside of UIKit, such as in your Xcode project, to make your application run smoothly. For example, this document discusses the configuration metadata that needs to accompany every application.

A good starting point for learning about application development is to learn a little about the environment in which applications must run. You can find this information in [“The Application Runtime Environment”](#) (page 13). After that, you should proceed to [“The Core Application Design”](#) (page 23) to learn about the core objects and behaviors (such as multitasking) found in an iOS application.

Supporting Common Application Behaviors

There are a handful of common behaviors that iOS applications can implement. These features are independent of the type of the application you are creating and are more closely related to the design goals you have for that application. For example, both games and productivity applications can be launched into a landscape orientation. And all applications need to be responsive to low-memory warnings coming from the system.

Deciding which behaviors you want to support is important to consider when designing your application. The amount of work required to implement each behavior is usually not large but such a feature might inspire you to rework part of your design to take better advantage of it.

- You can find steps for how to implement many common application behaviors in [“Implementing Standard Application Behaviors”](#) (page 47).
- If your application has specific needs that require it to run in the background, you should read [“Executing Code in the Background”](#) (page 63) for guidance on examples of how to implement your application.
- For information on how to write iOS applications to support devices with high resolution screens, see [“Supporting High-Resolution Screens”](#) (page 73).
- For information on how to support user-configurable preferences, read [“Implementing Application Preferences”](#) (page 81).

Meeting the App Store and System Requirements

Configuration of your application’s information property list file (`Info.plist`) and bundle are essential steps of the development process. The `Info.plist` file contains crucial information about your application’s configuration and supported features. The system relies heavily on this file to obtain information about your application and the location of key resource files needed to launch it. Also, bundles provide the fundamental structure for organizing your application’s resources and localized content. Knowing where to put things is important to building a running application.

To learn about the structure of iOS applications, and the steps you must take to configure them, read [“Build-Time Configuration Details”](#) (page 93).

Tuning Performance for the Underlying Device

In iOS, good application performance is particularly important and can mean the difference between success and failure. If your application is slow or consumes resources that prevent other applications from running smoothly, users are unlikely to want to buy it. And because resources such as memory are more constrained on iOS-based devices, it is imperative that you factor in system constraints to your design.

Power usage is a particularly important area of performance tuning when it comes to iOS applications. Many features require the system to enable specific bits of hardware. Disabling features that you are not using at the moment gives the system the opportunity to power down the associated hardware and extend battery life.

For information about techniques for improving performance and managing power usage, see [“Tuning for Performance and Responsiveness”](#) (page 107).

See Also

After reading this document, you should also consult the following documents for information about how to implement different parts of your application.

- For information about user interface design and how to create effective applications using iOS, see *iPhone Human Interface Guidelines* and *iPad Human Interface Guidelines*.
- For information about handling touch and motion-related events, see *Event Handling Guide for iOS*.
- For information about how to manage your application’s user interface, see *View Controller Programming Guide for iOS*.
- For information about windows, views, and drawing custom content, see *View Programming Guide for iOS*.
- For information about the text management facilities, see *Text and Web Programming Guide for iOS*.
- For information about how to incorporate audio and video into your applications, see *Multimedia Programming Guide*.
- For general information about all iOS technologies, see *iOS Technology Overview*.

The Application Runtime Environment

The runtime environment of iOS is designed for the fast and secure execution of programs. The following sections describe the key aspects of this runtime environment and provide guidance on how applications can best operate within it.

Fast Launch, Short Use

The strength of iOS-based devices is their immediacy. A typical user pulls a device out of a pocket or bag and uses it for a few seconds, or maybe a few minutes, before putting it away again. The user might be taking a phone call, looking up a contact, changing the current song, or getting some piece of information during that time. Your own applications need to reflect that sense of immediacy by launching and getting up and running quickly. If your application takes a long time to launch, the user may be less inclined to use it.

In iOS, the user interacts with only one application at a time. Thus, as each new application is launched, the previous application's user interface goes away. Prior to iOS 4, this meant that the application itself was quit and removed from memory. However, in iOS 4 and later, quitting an application causes it to move into the background. Applications remain in the background until they are launched again or until the system needs to reclaim the memory they are using.

The ability to run in the background means that relaunching an application is much faster. When it moves to the background, an application's objects remain in memory and in most cases the application itself enters a suspended state of execution. When the user relaunches the application, it simply resumes from the exact point where it was without having to reload its user interface and restore itself to its previous state.

Even though applications can run in the background, they are not guaranteed to do so forever. As memory becomes constrained, the system purges applications that have not been launched recently. Because this can happen at any time, and in some cases with no notice, your application should still save information about your application's current state in addition to any unsaved data when it moves to the background. If it ever needs to be relaunched, it can use this information to restore itself to the previous state and make it look like it was always running. Doing so provides a more consistent user experience by putting the user right back where they were when they last used your application.

Specialized System Behaviors

For the most part, iOS has the same features and behaves in the same way as Mac OS X. However, there are places where the behavior of iOS differs from Mac OS X.

The Virtual Memory System

To manage program memory, iOS uses essentially the same virtual memory system found in Mac OS X. In iOS, each program still has its own virtual address space, but (unlike Mac OS X) its usable virtual memory is constrained by the amount of physical memory available. This is because iOS does not write volatile pages to disk when memory gets full. Instead, the virtual memory system frees up volatile memory, as needed, to make sure the running application has the space it needs. It does this by removing memory pages that are not being used and that contain read-only contents, such as code pages. Such pages can always be loaded back into memory later if they are needed again.

If memory continues to be constrained, the system may also send notifications to the running applications, asking them to free up additional memory. All applications should respond to this notification and do their part to help relieve the memory pressure. For information on how to handle such notifications in your application, see [“Observing Low-Memory Warnings”](#) (page 47).

The Automatic Sleep Timer

One way iOS attempts to save power is through the automatic sleep timer. If the system does not detect touch events for an extended period of time, it dims the screen initially and eventually turns it off altogether. Although most developers should leave this timer on, game developers and developers whose applications do not use touch inputs can disable this timer to prevent the screen from dimming while their application is running. To disable the timer, set the `idleTimerDisabled` property of the shared `UIApplication` object to `YES`.

Because it results in greater power consumption, disabling the sleep timer should be avoided at all costs. The only applications that should consider using it are mapping applications, games, or applications that do not rely on touch inputs but do need to display visual content on the device’s screen. Audio applications do not need to disable the timer because audio content continues to play even after the screen dims. If you do disable the timer, be sure to reenable it as soon as possible to give the system the option to conserve more power. For additional tips on how to save power in your application, see [“Reduce Power Consumption”](#) (page 110).

Multitasking Support

In iOS 4 and later, applications can perform tasks while running in the background. When the user quits an application, instead of its process being terminated, the application is moved to the background. Shortly after moving to the background, most applications are suspended so that they do not run and consume additional power. However, applications that need to continue running can ask the system for the execution time to do so.

Regardless of whether an application is running in the background or suspended, the fact that the application is still in memory means that relaunching the application takes much less time. An application’s objects (including its windows and views) normally remain in memory and therefore do not need to be recreated when the application is subsequently relaunched by the user. However, if memory becomes constrained, the system may purge background applications to make more room for the foreground application. To appear consistent with the application remaining in the background at all times, an application should store information about its current state when it moves to the background and be able to restore itself to that state upon a subsequent relaunch.

For an overview of multitasking and what you need to do to support it, see [“Multitasking”](#) (page 36).

Security

An important job of iOS is to ensure the security of the user's device and the applications running on it. To this end, iOS implements several features to ensure the integrity of the user's data and to ensure that applications do not interfere with one another or the system.

The Application Sandbox

For security reasons, iOS restricts an application (including its preferences and data) to a unique location in the file system. This restriction is part of the security feature known as the application's "sandbox." The **sandbox** is a set of fine-grained controls limiting an application's access to files, preferences, network resources, hardware, and so on. In iOS, an application and its data reside in a secure location that no other application can access. When an application is installed, the system computes a unique opaque identifier for the application. Using a root application directory and this identifier, the system constructs a path to the application's home directory. Thus an application's home directory could be depicted as having the following structure:

/ApplicationRoot/ApplicationID/

During the installation process, the system creates the application's home directory and several key subdirectories, configures the application sandbox, and copies the application bundle to the home directory. The use of a unique location for each application and its data simplifies backup-and-restore operations, application updates, and uninstallation. For more information about the application-specific directories created for each application, see ["A Few Important Application Directories"](#) (page 16). For information about application updates and backup-and-restore operations, see ["Backup and Restore"](#) (page 18).

Important: The sandbox limits the damage an attacker can cause to other applications and to the system, but it cannot prevent attacks from happening. In other words, the sandbox does not protect your application from direct attacks by malicious entities. For example, if there is an exploitable buffer overflow in your input-handling code and you fail to validate user input, an attacker might still be able to crash your program or use it to execute the attacker's code.

File Protection

In iOS 4 and later, applications can use file protection to encrypt files and make them inaccessible when the user's device is locked. File protection takes advantage of built-in encryption hardware on specific devices (such as the iPhone 3GS) to add a level of security for applications that work with sensitive data. Protected files are stored on disk in an encrypted format at all times. While the user's device is locked, not even the owning application can access the data in the encrypted files. The user must explicitly unlock the device (by entering the appropriate passcode) before the application can retrieve the decrypted data from the files.

Protecting files on a device requires several steps:

- The file system on the user's device must be formatted to support file protection. For existing devices, this requires the user to reformat the device and restore any content from a backup.
- The user's device must have the passcode lock setting enabled and a valid passcode set.

- Applications must designate which data files need to be protected and assign the appropriate metadata attributes to them; see [“Marking a File as Protected”](#) (page 51).
- Applications must respond appropriately to situations where a file may become locked; see [“Determining the Availability of Protected Files”](#) (page 52).

It is up to you to decide which files your application should mark as protected. Applications must enable file protection on a file-by-file basis, and once enabled those protections cannot be removed. Also, applications should be prepared to deal with situations where data files should be protected but are not currently. This could happen if the user restored a device from an earlier backup where file protections were not yet added.

For more information about implementing support for file protection in your application, see [“Working with Protected Files”](#) (page 51).

Keychain Data

A keychain is a secure, encrypted container for passwords and other secrets. The keychain data for an application is stored outside of the application sandbox. If an application is uninstalled, that data is automatically removed. When the user backs up application data using iTunes, the keychain data is also backed up. However, it can only be restored to the device from which the backup was made. An upgrade of an application does not affect its keychain data.

For more on the iOS keychain, see [“Keychain Services Concepts”](#) in *Keychain Services Programming Guide*.

The File System

Your application and any files it creates share space with the user’s media and personal files on the flash-based memory. An application can access files using the local file system, which behaves like any other file system and is accessed using standard system routines. The following sections describe several things you should be aware of when accessing the local file system.

A Few Important Application Directories

For security purposes, an application has only a few locations in which it can write its data and preferences. When an application is installed on a device, a home directory is created for the application. Table 1-1 lists some of the important subdirectories inside the home directory that you might need to access. This table describes the intended usage and access restrictions for each directory and points out whether the directory’s contents are backed up by iTunes. For more information about the application home directory itself, see [“The Application Sandbox”](#) (page 15).

Table 1-1 Directories of an iOS application

Directory	Description
<code><Application_Home>/AppName.app</code>	<p>This is the bundle directory containing the application itself. Because an application must be signed, you must not make changes to the contents of this directory at runtime. Doing so may prevent your application from launching later.</p> <p>In iOS 2.1 and later, the contents of this directory are not backed up by iTunes. However, iTunes does perform an initial sync of any applications purchased from the App Store.</p>
<code><Application_Home>/Documents/</code>	<p>Use this directory to store user documents and application data files. The contents of this directory can be made available to the user through file sharing, which is described in “Sharing Files with the User’s Desktop Computer” (page 18).</p> <p>The contents of this directory are backed up by iTunes.</p>
<code><Application_Home>/Library/</code>	<p>This directory is the top-level directory for application-specific files. You typically put files in one of the standard subdirectories but you can create custom subdirectories too. (For information on how to get references to the standard subdirectories, see “Getting Paths to Application Directories” (page 49).) You should not use this directory for user data files.</p> <p>The contents of this directory (with the exception of the <code>Caches</code> subdirectory) are backed up by iTunes.</p>
<code><Application_Home>/Library/Preferences</code>	<p>This directory contains application-specific preference files. You should not create preferences files directly but should instead use the <code>NSUserDefaults</code> class or <code>CFPreferences</code> API to get and set application preferences; see also “Adding the Settings Bundle” (page 86).</p> <p>The contents of this directory are backed up by iTunes.</p>
<code><Application_Home>/Library/Caches</code>	<p>Use this directory to write any application-specific support files that you want to persist between launches of the application or during application updates. Your application is generally responsible for adding and removing these files. It should also be able to recreate these files as needed because iTunes removes them during a full restore of the device.</p> <p>In iOS 2.2 and later, the contents of this directory are not backed up by iTunes.</p>

Directory	Description
<code><Application_Home>/tmp/</code>	<p>Use this directory to write temporary files that you do not need to persist between launches of your application. Your application should remove files from this directory when it determines they are no longer needed. (The system may also purge lingering files from this directory when your application is not running.)</p> <p>In iOS 2.1 and later, the contents of this directory are not backed up by iTunes.</p>

For information about how to get the path of specific directories, see [“Getting Paths to Application Directories”](#) (page 49). For detailed information about which application directories are backed up, see [“What Is Backed Up?”](#) (page 18).

A Case-Sensitive File System

The file system for iOS-based devices is case sensitive. Whenever you work with filenames, you should be sure that the case matches exactly or your code may be unable to open or access the file.

Sharing Files with the User’s Desktop Computer

Applications that want to make user data files accessible can do so using application file sharing. File sharing enables the sharing of files between your application and the user’s desktop computer only. It does not allow your application to share files with other applications on the same device. To share data and files between applications, use the pasteboard or a document interaction controller object.

For information on how to support file sharing in your application, see [“Sharing Files with the User”](#) (page 51).

Backup and Restore

The iTunes application automatically handles the backup and restoration of user data in appropriate situations. However, applications need to know where to put files in order to ensure that they are backed up and restored (or not) as the case may be.

What Is Backed Up?

You do not have to prepare your application in any way for backup and restore operations. In iOS 2.2 and later, when a device is plugged into a computer and synced, iTunes performs an incremental backup of all files, except for those in the following directories:

- `<Application_Home>/AppName.app`
- `<Application_Home>/Library/Caches`

- `<Application_Home>/tmp`

Although iTunes does back up the application bundle itself, it does not do this during every sync operation. Applications purchased from the App Store on the device are backed up when that device is next synced with iTunes. Applications are not backed up during subsequent sync operations though unless the application bundle itself has changed (because the application was updated, for example).

To prevent the syncing process from taking a long time, you should be selective about where you place files inside your application's home directory. The `<Application_Home>/Documents` directory should be used to store user documents and application data files. Files used to store temporary data should be placed inside the `Application_Home/tmp` directory and deleted by your application when they are no longer needed. If your application creates data files that can be used during subsequent launches, but which do not need to be backed up, it should place those files in the `Application_Home/Library/Caches` directory.

Note: If your application creates large data files or files that change frequently, you should consider storing them in the `Application_Home/Library/Caches` directory and not in the `<Application_Home>/Documents` directory. Backing up large data files can slow down the backup process significantly. The same is true for files that change regularly (and therefore must be backed up frequently). Placing these files in the `Caches` directory prevents them from being backed up (in iOS 2.2 and later) during every sync operation.

For additional guidance about how you should use the directories in your application, see [Table 1-1](#) (page 17).

Files Saved During Application Updates

When a user downloads your application update, iTunes installs the update in a new application directory. It then moves the user's data files from the old installation over to the new application directory before deleting the old installation. Files in the following directories are guaranteed to be preserved during the update process:

- `<Application_Home>/Documents`
- `<Application_Home>/Library`

Although files in other user directories may also be moved over, you should not rely on their being present after an update.

The Simulator

The iPhone Simulator is a tool you can use to test your applications before deploying them to the App Store. The simulator provides a runtime environment that is close to, but not identical to, the one found on an actual device. Many of the restrictions that occur on devices, such as the lack of support for paging to disk, do not exist in the simulator. Also, some technologies such as OpenGL ES may not behave the same way in the simulator as they would on a device.

For more information about the simulator and its capabilities, see "Using iPhone Simulator" in *iOS Development Guide*.

Determining the Available Hardware Support

Applications designed for iOS must be able to run on devices with different hardware features. Although features such as the accelerometers and Wi-Fi networking are available on all devices, some devices may not include a camera or GPS hardware. If your application requires such features, you should always notify the user of that fact before they purchase the application. For features that are not required, but which you might want to support when present, you must check to see if the feature is available before trying to use it.

Important: If a feature absolutely must be present in order for your application to run, configure the `UIRequiredDeviceCapabilities` key in your application's `Info.plist` file accordingly. This key prevents users from installing applications that require features that are not present on a device. You should not include a key, however, if your application can function with or without the given feature. For more information about configuring this key, see [“Declaring the Required Device Capabilities”](#) (page 98).

Table 1-2 lists the techniques for determining if certain types of hardware are available. You should use these techniques in cases where your application can function without the feature but still uses it when present.

Table 1-2 Identifying available hardware features

Feature	Options
To determine if multitasking is available...	Get the value of the <code>multitaskingSupported</code> property in the <code>UIDevice</code> class. For more information about determining the availability of multitasking, see “Determining Whether Multitasking Support is Available” (page 63).
To determine if you should configure your interface for an iPad-sized screen or an iPhone-sized screen...	Use the <code>userInterfaceIdiom</code> property of the <code>UIDevice</code> class. This property is applicable only to universal applications that support different layouts based on whether the content is intended for iPad versus iPhone and iPod touch. For more information on implementing a universal application, see “Starting Your Project” in <i>iPad Programming Guide</i> .
To determine if an external screen is attached...	Get the value of the <code>screens</code> property in the <code>UIScreen</code> class. If the array contains more than one screen object, one of the objects corresponds to the main screen and the other corresponds to an external screen. For more information about using additional screens, see <i>UIScreen Class Reference</i> .
To determine if hardware-level disk encryption is available...	Get the value of the <code>protectedDataAvailable</code> property in the shared <code>UIApplication</code> object. For more information on encrypting on-disk content, see “Working with Protected Files” (page 51).
To determine if the network is available...	Use the reachability interfaces of the System Configuration framework to determine the current network connectivity. For an example of how to use the System Configuration framework, see <i>Reachability</i> .
To determine if the still camera is available...	Use the <code>isSourceTypeAvailable:</code> method of the <code>UIImagePickerController</code> class to determine if a camera is available. For more information, see <i>Device Features Programming Guide</i> .

Feature	Options
To determine if the device can capture video...	Use the <code>isSourceTypeAvailable:</code> method of the <code>UIImagePickerController</code> class to determine if a camera is available and then use the <code>availableMediaTypesForSourceType:</code> method to request the types for the <code>UIImagePickerControllerSourceTypeCamera</code> source. If the returned array contains the <code>kUTTypeMovie</code> key, video capture is available. For more information, see <i>Device Features Programming Guide</i> .
To determine if audio input (a microphone) is available...	In iOS 3 and later, use the <code>AVAudioSession</code> class to determine if audio input is available. This class accounts for many different sources of audio input on iOS-based devices, including built-in microphones, headset jacks, and connected accessories. For more information, see <i>AVAudioSession Class Reference</i> .
To determine if GPS hardware is present...	Specify a high accuracy level when configuring the <code>CLLocationManager</code> object to deliver location updates to your application. The Core Location framework does not provide direct information about the availability of specific hardware but instead uses accuracy values to provide you with the data you need. If the accuracy reported in subsequent location events is insufficient, you can let the user know. For more information, see <i>Location Awareness Programming Guide</i> .
To determine if a specific hardware accessory is available...	Use the classes of the External Accessory framework to find the appropriate accessory object and connect to it. For more information, see <i>External Accessory Programming Topics</i> .
To get the current battery level of the device...	Use the <code>batteryLevel</code> property of the <code>UIDevice</code> class. For more information about this class, see <i>UIDevice Class Reference</i> .
To get the state of the proximity sensor...	Use the <code>proximityState</code> property of the <code>UIDevice</code> class. Not all devices have proximity sensors, so you should also check the <code>proximityMonitoringEnabled</code> property to determine if this sensor is available. For more information about this class, see <i>UIDevice Class Reference</i> .

To obtain general information about device- or application-level features, use the methods and properties of the `UIDevice` and `UIApplication` classes. For more information about these classes, see *UIDevice Class Reference* and *UIApplication Class Reference*.

The Core Application Design

Every iOS application is built using the UIKit framework and therefore has essentially the same core architecture. UIKit provides the key objects needed to run the application, to coordinate the handling of user input, and to display content on the screen. Where applications deviate from one another is in how they configure these default objects and also where they incorporate custom objects to augment their application's user interface and behavior.

There are many interactions that occur between the system and a running application, and many of these interactions are handled automatically by the UIKit infrastructure. However, there are times when your application needs to be aware of the events coming from the system. For example, when the user quits your application, you need to save any relevant data before it goes away. For these situations, UIKit provides hooks that your custom code can use to provide the needed behavior.

Fundamental Design Patterns

The design of the UIKit framework incorporates many of the design patterns found in Cocoa applications in Mac OS X. Understanding these design patterns is crucial to creating iOS applications, so it is worth taking a few moments to learn about them. Table 2-1 provides a brief overview of these design patterns.

Table 2-1 Design patterns used by iOS applications

Design pattern	Description
Model-View-Controller	The Model-View-Controller (MVC) design pattern is a way of dividing your code into independent functional areas. The model portion defines your application's underlying data engine and is responsible for maintaining the integrity of that data. The view portion defines the user interface for your application and has no explicit knowledge of the origin of data displayed in that interface. The controller portion acts as a bridge between the model and view and facilitates updates between them.
Block objects	Block objects are a convenient way to encapsulate code and local stack variables in a form that can be executed later. Support for block objects is available in iOS 4 and later, where they are often used in place of delegates to act as callbacks for asynchronous tasks.
Delegation	The delegation design pattern is a way of modifying complex objects without subclassing them. Instead of subclassing, you use the complex object as is and put any custom code for modifying the behavior of that object inside a separate object, which is referred to as the <i>delegate object</i> . At predefined times, the complex object then calls the methods of the delegate object to give it a chance to run its custom code.

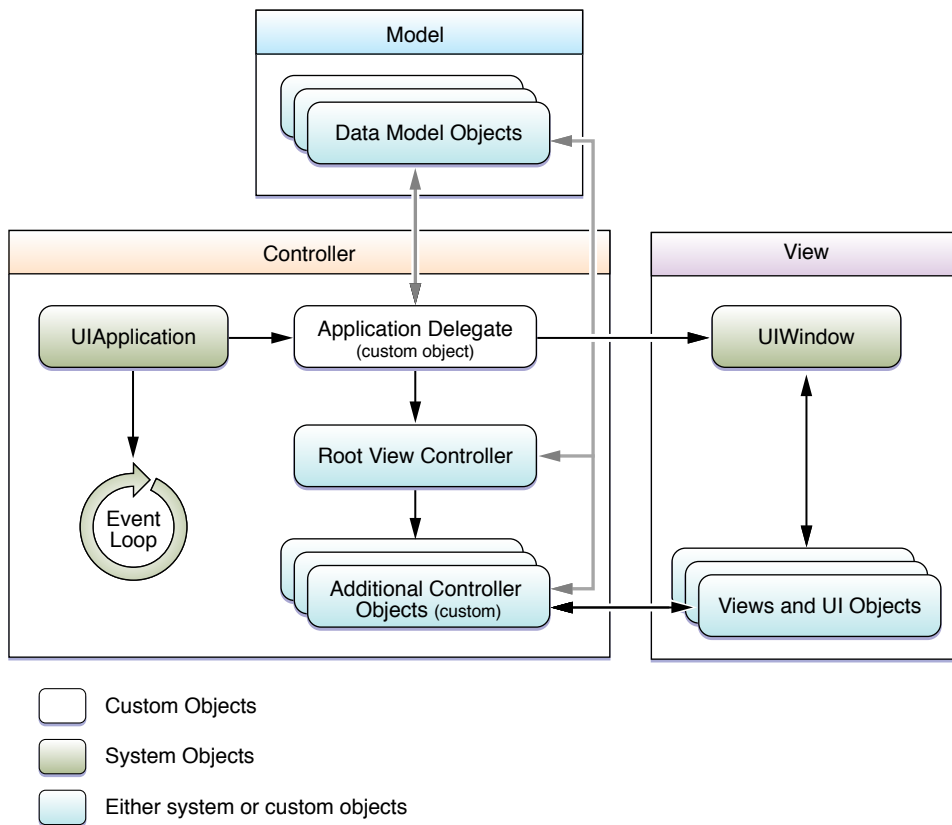
Design pattern	Description
Target-action	Controls use the target-action design pattern to notify your application of user interactions. When the user interacts with a control in a predefined way (such as by tapping a button), the control sends a message (the action) to an object you specify (the target). Upon receiving the action message, the target object can then respond in an appropriate manner (such as by updating application state in response to the button push).
Managed memory model	The Objective-C language uses a reference-counted scheme for determining when to release objects from memory. When an object is first created, it is given a reference count of 1. Other objects can then use the <code>retain</code> , <code>release</code> , or <code>autorelease</code> methods of the object to increase and decrease that reference count appropriately. When an object's reference count reaches 0, the Objective-C runtime calls the object's cleanup routines and then deallocates it.
Threads and concurrent programming	All versions of iOS support the creation of operation objects and secondary threads. In iOS 4 and later, applications can also use Grand Central Dispatch (GCD) as the underlying mechanism for executing tasks concurrently. For more information about concurrency and the technologies available for implementing it, see <i>Concurrency Programming Guide</i> .

For a more thorough discussion of these design patterns, see *Cocoa Fundamentals Guide*.

The Core Application Objects

From the time your application is launched by the user, to the time it exits, the UIKit framework manages most of the application's key infrastructure. An iOS application receives events continuously from the system and must respond to those events. Receiving the events is the job of the `UIApplication` object, but responding to the events is the responsibility of your custom code. To understand where you need to respond to events, though, it helps to understand a little about the overall life cycle and event cycles of an iOS application. The following sections describe these cycles and also provide a summary of some of the key design patterns used throughout the development of iOS applications.

In addition to showing the `UIApplication` object, Figure 2-1 shows the objects that are most commonly found in an iOS application, and Table 2-2 describes the roles of each of these types of objects.

Figure 2-1 Key objects in an iOS application**Table 2-2** The role of objects in an iOS application

Object	Description
<code>UIApplication</code> object	The <code>UIApplication</code> object manages the application event loop and coordinates other high-level behaviors for your application. You use this object as is, mostly to configure various aspects of your application's appearance. Your custom application-level code resides in your application delegate object, which works in tandem with this object.
Application delegate object	The application delegate is a custom object that you provide at application launch time, usually by embedding it in your application's main nib file. The primary job of this object is to initialize the application and present its window onscreen. The <code>UIApplication</code> object also notifies this object when specific application-level events occur, such as when the application needs to be interrupted (because of an incoming message) or suspended (because the user tapped the Home button). For more information about this object, see “The Application Delegate” (page 29).

Object	Description
Data model objects	Data model objects store your application's content and are therefore custom to your application. For example, a banking application might store information about financial transactions, while a painting application might store an image object or even the sequence of drawing commands that led to the creation of that image. In the latter case, an image object is still a data object because it is just a container for the image data. The actual rendering of that image still takes place elsewhere in your application.
View controller objects	<p>View controller objects manage the presentation of a screen's worth of content for your application. Typically, this involves loading the views responsible for presenting your content, creating any additional generic controllers to manage portions of the content, and coordinating the interactions with your application's data model objects.</p> <p>The <code>UIViewController</code> class is the base class for all view controller objects. It provides default functionality for standard system behaviors such as animating the appearance of views and handling rotations. More specific view controller subclasses provide additional behaviors, such as displaying system-specific screens or interface types.</p> <p>For detailed information about how to use view controllers, see <i>View Controller Programming Guide for iOS</i>.</p>
<code>UIWindow</code> object	<p>A <code>UIWindow</code> object manages the drawing surface for your application. Most applications have only one window. An application changes the content of that window by presenting a new set of views using a view controller object.</p> <p>In addition to hosting views, windows are also responsible for delivering events to those views and to their managing view controllers.</p>
Views, controls, and layers	<p>Views and controls provide the visual representation of your application's content. A view is an object that draws some content in a designated rectangular area and responds to events within that area. Controls are a specialized type of view responsible for implementing familiar interface objects such as buttons, text fields, and toggle switches.</p> <p>The UIKit framework provides standard views for presenting many different types of content. You can also define your own custom views by subclassing <code>UIView</code> (or its descendants) directly.</p> <p>In addition to incorporating views and controls, applications can also incorporate Core Animation layers into their view and control hierarchies. Layer objects are actually data objects that represent visual content. Views use layer objects intensively behind the scenes to render their content. You can also add custom layer objects to your interface to implement complex animations and other types of sophisticated visual effects.</p>

The objects in your application form a complex ecosystem, the specifics of which are what define your application. As you can see from [Figure 2-1](#) (page 25), most of the objects in an application are either partially or wholly customizable. Fortunately, an application that builds on top of existing UIKit classes receives a significant amount of infrastructure for free. All you have to do is understand the specific points where you can override or augment the default behavior to implement the customizations you need. The remainder of

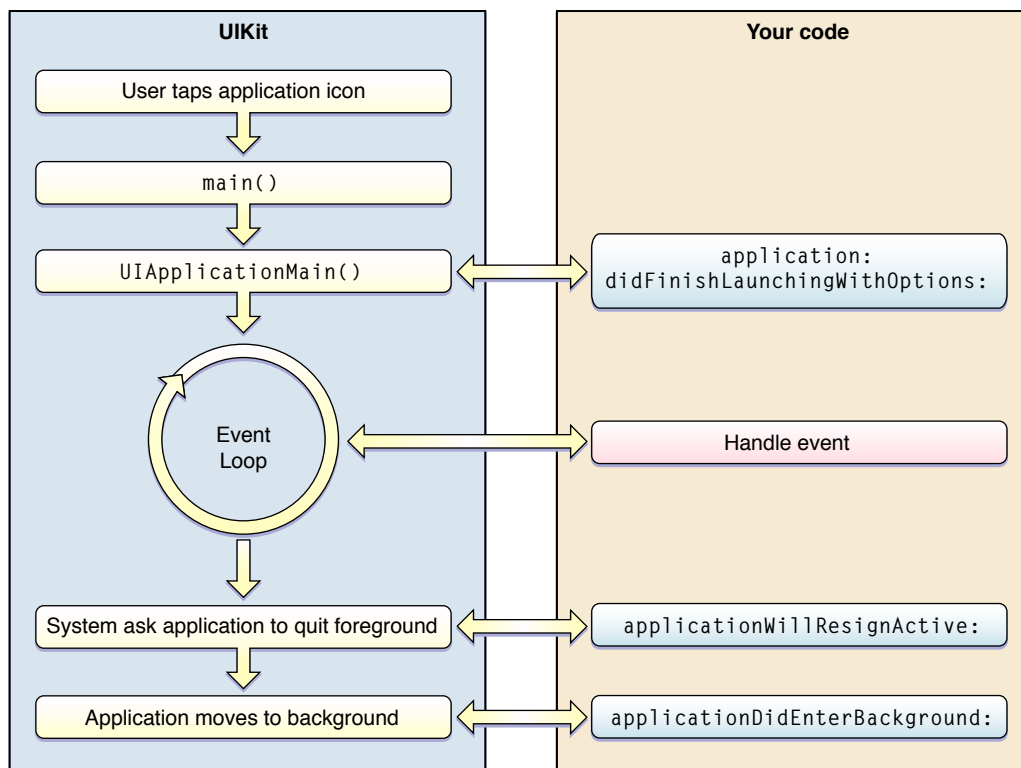
this chapter focuses on the override points that encompass your application's overall behavior and interactions with the system. It also points you to additional documents where you can find out more about specific types of interactions.

The Application Life Cycle

The application life cycle constitutes the sequence of events that occurs between the launch and termination of your application. In iOS, the user launches your application by tapping its icon on the Home screen. Shortly after the tap occurs, the system displays some transitional graphics and proceeds to launch your application by calling its `main` function. From this point on, the bulk of the initialization work is handed over to UIKit, which loads the application's user interface and readies its event loop.

Figure 2-2 depicts the simplified life cycle of an iOS application. This diagram shows the sequence of events that occur from the time the application starts up to the time it quits. At key points in the application's life, UIKit sends messages to the application delegate object to let it know what is happening. During the event loop, UIKit dispatches events to your application's custom event handlers.

Figure 2-2 Application life cycle



Prior to iOS 4, only one application at a time was ever allowed to run. As a result, before a new application could be launched, the previous application had to be terminated. This resulted in an application life cycle in which the application was always launched from scratch and then removed from memory at quit time. In iOS 4 and later, applications now remain resident in memory by default after they are quit. This means that subsequent launches of the application may not always involve starting the application from scratch. It also means that you need to design your application code to handle several different state transitions.

The Main Function

Like any C-based application, the main entry point for an iOS application at launch time is the `main` function. In an iOS application, the `main` function is used only minimally. Its main job is to hand control off to the UIKit framework. Therefore, any new project you create in Xcode comes with a default main function like the one shown in Listing 2-1. With few exceptions, you should never change the implementation of this function.

Listing 2-1 The `main` function of an iOS application

```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Note: An autorelease pool is used in memory management. It is a Cocoa mechanism used to defer the release of objects created during a functional block of code. For more information about autorelease pools, see *Memory Management Programming Guide*. For specific memory-management guidelines related to autorelease pools in iOS applications, see [“Allocate Memory Wisely”](#) (page 108).

The `UIApplicationMain` function at the heart of your application’s `main` function takes four parameters and uses them to initialize the application. Although you should never have to change the default values passed into this function, it is worth explaining their purpose in terms of starting the application.

- The `argc` and `argv` parameters contain any launch-time arguments passed to the application from the system. These arguments are parsed by the UIKit infrastructure and can otherwise be ignored.
- The third parameter identifies the name of the application’s principal class. This is the class responsible for running the application. Specifying `nil` causes UIKit to use the `UIApplication` class, which is recommended.
- The fourth parameter identifies the class of the application delegate. The application delegate is responsible for managing the high-level interactions between the system and your custom code. Specifying `nil` tells UIKit that the application delegate object is located in the application’s main nib file (which is the case for applications built using the Xcode templates).

In addition to creating the application object and creating or loading the application delegate, the `UIApplicationMain` function also loads the application’s main nib file. All Xcode projects have a main nib file by default, and this file typically contains the application’s window and the application delegate object. UIKit obtains the name of the main nib file by looking at the value in the `NSMainNibFile` key in your application’s `Info.plist` file. Although you should rarely need to do so, you can designate a new main nib file for your application by changing the value of this key before building your project. For more information about the `Info.plist` file and how you use it to configure your application, see [“The Information Property List”](#) (page 95).

The Application Delegate

Monitoring the high-level behavior of your application is the responsibility of the application delegate object, which is a custom object that you provide. Delegation is a mechanism used to avoid subclassing complex UIKit objects, such as the default `UIApplication` object. Instead of subclassing and overriding methods, you use the complex object unmodified and put your custom code inside the delegate object. As interesting events occur, the complex object sends messages to your delegate object. You can use these “hooks” to execute your custom code and implement the behavior you need.

Important: The delegate design pattern is intended to save you time and effort when creating applications and is therefore a very important pattern to understand. For an overview of the key design patterns used by iOS applications, see [“Fundamental Design Patterns”](#) (page 23). For a more detailed description of delegation and other UIKit design patterns, see *Cocoa Fundamentals Guide*.

The application delegate object is responsible for handling several critical system messages and *must* be present in every iOS application. The object can be an instance of any class you like, as long as it adopts the `UIApplicationDelegate` protocol. The methods of this protocol define the hooks into the application life cycle and are your way of implementing custom behavior.

For additional information about the methods of the `UIApplicationDelegate` protocol, see *UIApplicationDelegate Protocol Reference*.

Understanding an Application’s States and Transitions

Applications running in iOS 4 and later can be in one of several different states at any given time. Table 2-3 lists the potential runtime states of an application in iOS 4 and later.

Table 2-3 Application states

State	Description
Not running	The application has not been launched or was running but was terminated by the system.
Inactive	The application is running in the foreground but is currently not receiving events. (It may be executing other code though.) An application usually stays only briefly in this state as it transitions to a different state. The only time it stays inactive for any period of time is when the user locks the screen or the system prompts the user to respond to some event, such as an incoming phone call or SMS message.
Active	The application is running in the foreground and is receiving events.
Background	<p>The application is in the background and executing code. Most applications enter this state briefly on their way to being suspended. However, an application that requests extra execution time may remain in this state for a period of time. In addition, an application being launched directly into the background enters this state instead of the inactive state. For information about how to execute code while in the background, see “Executing Code in the Background” (page 63).</p> <p>The background state is only available in iOS 4 and later and on devices that support multitasking. If this state is not available, applications are terminated and moved to the not running state instead.</p>

State	Description
Suspended	<p>The application is in the background but is not executing code. The system moves application to this state automatically and at appropriate times. While suspended, an application is essentially freeze-dried in its current state and does not execute any code. During low-memory conditions, the system purges suspended applications without notice to make more space for the foreground application.</p> <p>The suspended state is only available in iOS 4 and later and on devices that support multitasking. If this state is not available, applications are terminated and moved to the not running state instead.</p>

The following sections describe the key state transitions in more detail and call out the typical behaviors your application should observe during the transition.

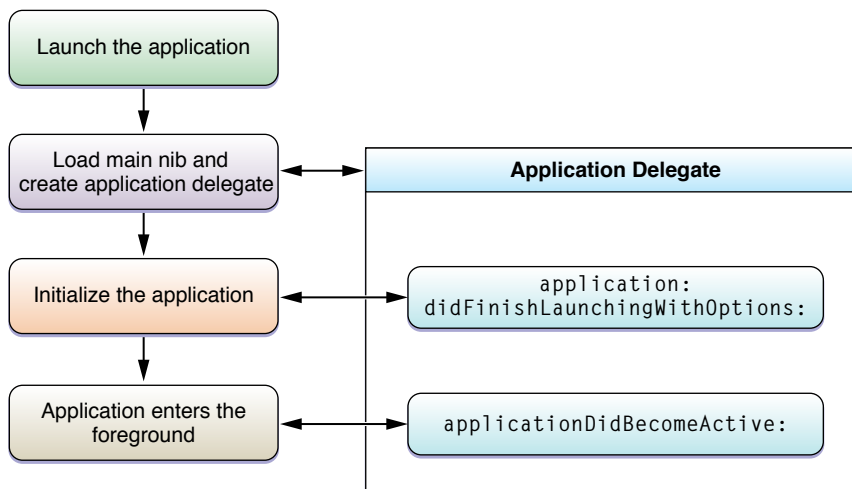
Launching the Application

At launch time, an application moves from the not running state to either the active or the background state. A launching application has to prepare itself to run and then check to see if the system has launched it in order to perform a specific task. During the initial startup sequence, the application calls its delegate's `application:didFinishLaunchingWithOptions:` method followed by either the `applicationDidBecomeActive:` or `applicationDidEnterBackground:` method (depending on whether it is transitioning to the foreground or background).

Note: Launching into the background state does not occur in iOS 3.x and earlier or on devices that do not support multitasking. Applications in those circumstances launch only into the active state.

Figure 2-3 shows the sequence of steps that occur when launching into the foreground. The sequence for launching into the background is the same except that the `applicationDidBecomeActive:` method is replaced by the `applicationDidEnterBackground:` method.

Figure 2-3 Launching into the active state



The application delegate's `application:didFinishLaunchingWithOptions:` method is responsible for doing most of the work at launch time and has the following responsibilities:

- Initialize the application's data structures.
- Load or create the application's initial window and views in a portrait orientation.

If the device is in a non-portrait orientation at launch time, you should still create your interface in a portrait orientation initially. After the `application:didFinishLaunchingWithOptions:` method returns, the application tells the window to rotate its contents to the correct orientation. The window then uses the normal orientation-change mechanism of its view controllers to make the rotation happen prior to becoming visible. Information about how interface orientation changes work is described in “Custom View Controllers” in *View Controller Programming Guide for iOS*.
- Check the contents of the launch options dictionary for information about why the application was launched and respond appropriately.
- Use any saved preferences or state information to restore the application to its previous runtime state.

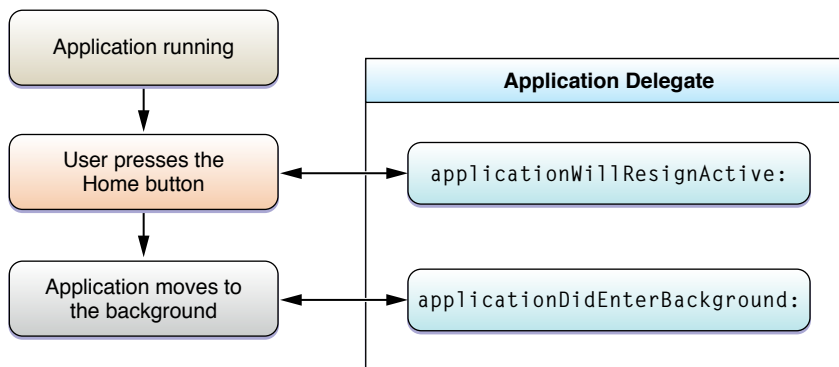
You should strive to make your `application:didFinishLaunchingWithOptions:` method as lightweight as possible. Although there are any number of custom initialization tasks you could perform in this method, if this method takes too long to complete those tasks, the system might terminate the application for being unresponsive. One way to make the method lightweight is to initiate tasks asynchronously or move any long-running tasks to secondary threads. This is especially important for network-based operations that could take an indeterminate amount of time to complete.

When your `application:didFinishLaunchingWithOptions:` method is called, the `applicationState` property of the `UIApplication` object is already set to the appropriate state for your application. If the property is set to `UIApplicationStateInactive`, your application is in the inactive state and is about to be moved to the foreground. If it is set to `UIApplicationStateBackground`, the application is about to be moved to the background. In either case, you can use this information to prepare your application appropriate for running.

Applications are launched into the background only as needed to handle an incoming background event. When launched into the background, an application generally has a limited amount of execution time and should therefore avoid doing work that is not immediately relevant to processing the background event. For example, you should avoid setting up your application's user interface. Instead, you should make a note that the interface needs to be configured and do that work when moving to the foreground later. For additional guidance about how to configure your application for background execution, see [“Being a Responsible, Multitasking-Aware Application”](#) (page 65).

Moving to the Background

When the user presses the Home button or the system launches another application, the foreground application transitions first to the inactive state and then to the background state. This results in calls to the application delegate's `applicationWillResignActive:` and `applicationDidEnterBackground:` methods, as shown in Figure 2-4. Most background applications move to the suspended state shortly after returning from the `applicationDidEnterBackground:` method. If your application requests more execution time or declares itself to support background execution, it is allowed to continue running after this method returns.

Figure 2-4 Moving from the foreground to the background

When your delegate's `applicationDidEnterBackground:` is called, your application has approximately five seconds to finish any lingering tasks and return. In practice, you should return from this method as quickly as possible. If the method does not return before time runs out (or does not request more execution time from the system), your application is terminated and purged from memory. Any tasks relating to adjusting your user interface must be performed before this method exits but other tasks should be moved to a concurrent dispatch queue or secondary thread as needed. Of course, you may need to request background execution time to ensure the code finishes executing before your application is suspended.

The `UIApplicationDidEnterBackgroundNotification` notification is also sent to notify interested parts of your application that it is entering the background. Objects in your application can use the default notification center to register for this notification.

You should use the `applicationDidEnterBackground:` method to save user data and any state information needed to restore your application in the event that it is subsequently purged from memory. When an application enters the background state, there is no guarantee that it will remain there indefinitely. If the device's memory continues to be constrained, the system purges background applications to make more room. If your application is suspended when that happens, it receives no notice that it is removed from memory. Thus, you need to save any data beforehand.

Important: All multitasking-aware applications should behave responsibly when moving to the background. This is true regardless of whether your application is suspended shortly after entering the background or continues running. Saving the user's data is one step that should always be taken, but there other guidelines that you should follow. For a list of these guidelines, see [“Being a Responsible, Multitasking-Aware Application”](#) (page 65).

When an application moves to the background, all of your core application objects remain in memory and are available for use while in the background. These objects include your custom objects and data structures plus your application's controller objects, windows, and views. However, the system does release many of the objects used behind-the-scenes to support your application. Specifically, the system does the following for background applications:

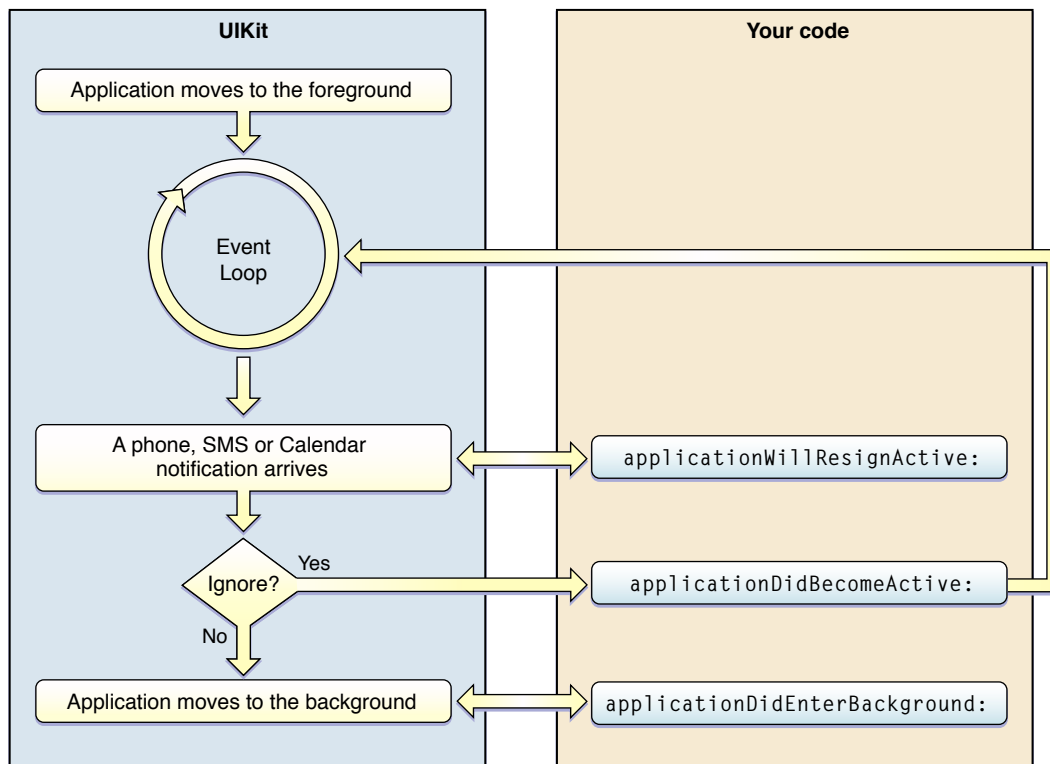
- It releases the backing store for all Core Animation layers, which prevents the contents of those layers from appearing onscreen but does not change the current layer properties.
- It releases any references to cached images. (If your application does not retain the images, they are subsequently removed from memory.)
- It releases some other system-managed data caches.

If the application is running in iOS 3.x and earlier or on a device that does not support multitasking, the application is terminated instead of being moved to the background. For more information about how to respond to the termination of your application, see “[Responding to Application Termination](#)” (page 35).

Responding to Interruptions

When the application is interrupted by an incoming phone call, SMS, or calendar notification, the application moves to the inactive state temporarily. It remains in this state until the user decides whether to accept or ignore the interruption. If the user ignores the interruption, the application is reactivated, at which time it can resume any services it stopped when it moved into the inactive state. If the user accepts the interruption, the application moves into the suspended state. Figure 2-5 shows the set of steps this process follows. The steps that follow describe this process in more detail.

Figure 2-5 Handling application interruptions



1. The system detects an incoming phone call or SMS message, or a calendar event occurs.
2. The system calls your application delegate's `applicationWillResignActive:` method. The system also disables the delivery of touch events to your application.

Interruptions amount to a temporary loss of control by your application. If such a loss of control might affect your application's behavior or cause a negative user experience, you should take appropriate steps in your delegate method to prevent that from happening. For example, if your application is a game, you should pause the game. You should also disable timers, throttle back your OpenGL frame rates (if using OpenGL), and generally put your application into a sleep state. While your application is in the inactive state, it continues to run but should not do any significant work.

3. The system displays an alert panel with information about the event. The user can choose to ignore the event or respond to it.
4. If the user ignores the event, the system calls your application delegate's `applicationDidBecomeActive:` method and resumes the delivery of touch events to your application.

You can use this delegate method to reenable timers, throttle up your OpenGL frame rates, and generally wake up your application from its sleep state. For games that are in a paused state, you should consider leaving the game in that state until the user is ready to resume play. For example, you might display an alert panel with controls to resume play.

5. If the user responds to the event instead of ignoring it, the system calls your application delegate's `applicationDidEnterBackground:` method. Your application should move to the background as usual, saving any user data or contextual information needed to restore your application to its current state later.

If the application is running in iOS 3.x or earlier or on a device that does not support multitasking, the application delegate's `applicationWillTerminate:` method is called instead of the `applicationDidEnterBackground:` method.

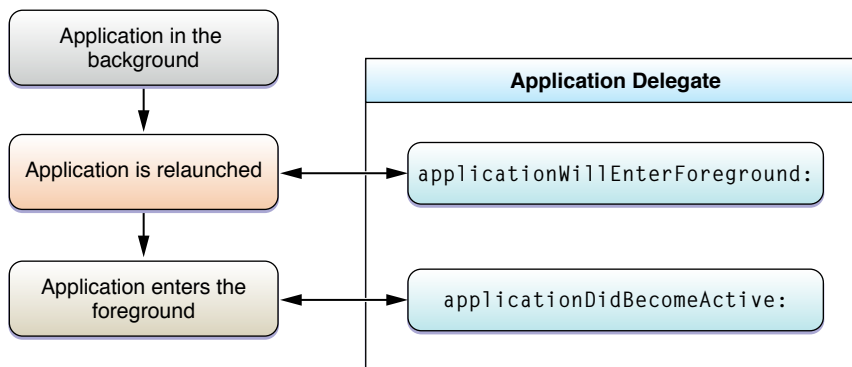
Depending on what the user does while responding to an interruption, the system may return to your application when that interruption ends. For example, if the user takes a phone call and then hangs up, the system relaunches your application. If, while on the call, the user goes back to the Home screen or launches another application, the system does not return to your application.

Important: When the user takes a call and then returns to your application while on the call, the height of the status bar grows to reflect the fact that the user is on a call. Similarly, when the user ends the call, the status bar height shrinks back to its regular size. Your application should be prepared for these changes in the status bar height and adjust its content area accordingly. View controllers handle this behavior for you automatically. If you lay out your user interface programmatically, however, you need to take the status bar height into account when laying out your views and implement the `layoutSubviews` method to handle dynamic layout changes.

If the user presses the Sleep/Wake button on a device while running your application, the system calls your application delegate's `applicationWillResignActive:` method, stops the delivery of touch events, and then puts the device to sleep. When the user wakes the device later, the system calls your application delegate's `applicationDidBecomeActive:` method and begins delivering events to the application again. While the device is asleep, foreground and background applications continue to run, but those applications should do as little work as possible in order to preserve battery life.

Resuming Foreground Execution

When the user launches an application that currently resides in the background, the system moves it to the inactive state and then to the active state. This results in calls to the `applicationWillEnterForeground:` and `applicationDidBecomeActive:` methods of the application delegate, as shown in Figure 2-6.

Figure 2-6 Transitioning from the background to the foreground

When moving to the foreground, your application should restart any services it stopped and generally prepare itself for handling events again.

Note: The `UIApplicationWillEnterForegroundNotification` notification is also available for tracking when your application reenters the foreground. Objects in your application can use the default notification center to register for this notification.

While an application is suspended, the system tracks and coalesces events that might have an impact on that application when it relaunches. As soon as your application is up and running again, the system delivers those events to it. For most of these events, your application's existing infrastructure should just respond appropriately. For example, if the device orientation changed, your application's view controllers would automatically update the interface orientation in an appropriate way. For more information about the types of events that are tracked by the system while your application is in the background, and the appropriate way to handle those events, see [“Responding to System Changes While in the Background”](#) (page 36).

Responding to Application Termination

Although applications are generally moved to the background and suspended, if any of the following conditions are true, your application is terminated and purged from memory instead of being moved to the background:

- The application is linked against iOS SDK 3.x or earlier.
- The application is deployed on a device running iOS 3.x or earlier.
- The current device does not support multitasking; see [“Determining Whether Multitasking Support is Available”](#) (page 63).
- The application includes the `UIApplicationExitsOnSuspend` key in its `Info.plist` file; see [“Opting Out of Background Execution”](#) (page 38).

If your application is running (either in the foreground or background) at termination time, the system calls your application delegate's `applicationWillTerminate:` method so that you can perform any required cleanup. You can use this method to save user data or application state information that you would use to restore your application to its current state on a subsequent launch. Your method implementation has approximately five seconds to perform any tasks and return. If it does not return in time, the application is forcibly terminated and removed from memory. The `applicationWillTerminate:` method is not called if your application is currently suspended.

Even if you develop your application using iOS SDK 4 and later, you must still be prepared for your application to be terminated. The user can terminate applications explicitly using the multitasking UI. In addition, if memory becomes constrained, the system might remove applications from memory in order to make more room. If your application is currently suspended, the system removes your application from memory without any notice. However, if your application is currently running in the background state (in other words, not suspended), the system calls the `applicationWillTerminate:` method of your application delegate. Your application cannot request additional background execution time from this method.

Multitasking

In iOS 4 and later, multiple applications may reside in memory and run simultaneously. Only one application runs in the foreground while all other applications reside in the background. Applications running in this environment must be designed to handle transitions between the foreground and background.

Checklist for Supporting Multitasking

Applications that support multitasking in iOS 4 and later should do the following:

- **(Required)** Respond appropriately to the state transitions that occur while running under multitasking. Applications need to observe these transitions in order to save state and tailor their behavior for foreground or background execution. Not handling these transitions properly could lead to data loss or improper behavior. For more information about the states and transitions, see [“Understanding an Application’s States and Transitions”](#) (page 29).
- **(Required)** Follow the guidelines for behavior when moving to the background. These guidelines are there to help your application behave correctly while in the background and in situations where your application might need to be terminated. For information about these guidelines, see [“Being a Responsible, Multitasking-Aware Application”](#) (page 65).
- **(Recommended)** Register for any notifications that report system changes your application needs. The system queues notifications while an application is suspended and delivers them once the application resumes execution so that it can make a smooth transition back to execution. For more information, see [“Responding to System Changes While in the Background”](#) (page 36).
- **(Optional)** If you want to do actual work while in the background, you need to request permission to continue running. For more information about the types of work you can perform, and how to request permission to do that work, see [“Executing Code in the Background”](#) (page 63).

If you do not want to support multitasking at all, you can opt out and elect to always have your application terminated and purged from memory at quit time. For information on how to do this, see [“Opting Out of Background Execution”](#) (page 38).

Responding to System Changes While in the Background

While an application is in the suspended state, it does not receive system-related events that might be of interest. However, the most relevant events are captured by the system and queued for later delivery to your application. To prevent your application from becoming overloaded with notifications when it resumes, the system coalesces events and delivers a single event (of each relevant type) corresponding to the net change since your application was suspended.

To understand how this might work in your application, consider an example. Suppose that at the time when your application is suspended, the device is in a portrait orientation. While the application is suspended, the user rotates the device to landscape left, upside down, and finally landscape right orientations before launching your application again. Upon resumption, your application would receive a single device orientation event indicating that the device changed to a landscape-right orientation. Of course, if your application uses view controllers, the orientation of those view controllers would be updated automatically by UIKit. Your application would need to respond only if it tracked device orientation changes explicitly.

Table 2-4 lists the events that are coalesced and delivered to your application. In most cases, the events are delivered in the form of a notification object. However, some events may be intercepted by a system framework and delivered to your application by another means. Unless otherwise noted, all events are delivered regardless of whether your application resumes in the foreground or background.

Table 2-4 Notifications delivered to waking applications

Event	Notification mechanism
Your code marks a view as dirty	Any calls to <code>setNeedsDisplay</code> or <code>setNeedsDisplayInRect:</code> on one of your views are coalesced and stored until your application resumes in the foreground. These events are not delivered to applications running in the background.
An accessory is connected or disconnected	<code>EAAccessoryDidConnectNotification</code> <code>EAAccessoryDidDisconnectNotification</code>
The device orientation changes	<code>UIDeviceOrientationDidChangeNotification</code> In addition to this notification, view controllers update their interface orientations automatically.
There is a significant time change	<code>UIApplicationSignificantTimeChangeNotification</code>
The battery level or battery state changes	<code>UIDeviceBatteryLevelDidChangeNotification</code> <code>UIDeviceBatteryStateDidChangeNotification</code>
The proximity state changes	<code>UIDeviceProximityStateDidChangeNotification</code>
The status of protected files changes	<code>UIApplicationProtectedDataWillBecomeUnavailable</code> <code>UIApplicationProtectedDataDidBecomeAvailable</code>
An external display is connected or disconnected	<code>UIScreenDidConnectNotification</code> <code>UIScreenDidDisconnectNotification</code>
The screen mode of a display changes	<code>UIScreenModeDidChangeNotification</code>
Preferences that your application exposes through the Settings application are changed	<code>NSUserDefaultsDidChangeNotification</code>
The current language or locale settings changes	<code>NSCurrentLocaleDidChangeNotification</code>

When your application resumes, any queued events are delivered via your application's main run loop. Because these events are queued right away, they are typically delivered before any touch events or other user input. Most applications should be able to handle these events quickly enough that they would not cause any noticeable lag when resumed. However, if your application appears sluggish in responding to user input when it is woken up, you might want to analyze your application using Instruments and see if your handler code is causing the delay.

Handling Locale Changes Gracefully

If the user changes the language or locale of the device while your application is suspended, the system notifies you of that change using the `NSCurrentLocaleDidChangeNotification` notification. You can use this notification to force updates to any views containing locale-sensitive information, such as dates, times, and numbers. Of course, you should also be careful to write your code in ways that might make it easy to update things easily:

- Use the `autoupdatingCurrentLocale` class method when retrieving `NSLocale` objects. This method returns a locale object that updates itself automatically in response to changes, so you never need to recreate it.
- Avoid caching `NSFormatter` objects. Date and number formatters must be recreated whenever the current locale information changes.

Responding to Changes in Your Application's Settings

If your application has settings that are managed by the Settings application, it should observe the `NSUserDefaultsDidChangeNotification` notification. Because the user can modify settings while your application is in the background, you can use this notification to respond to any important changes in those settings. For example, an email program would need to respond to changes in the user's mail account information. Failure to do so could have serious privacy and security implications. Specifically, the user might still be able to send email using the old account information, even if the account did not belong to that person.

Upon receiving the `NSUserDefaultsDidChangeNotification` notification, your application should reload any relevant settings and, if necessary, reset its user interface appropriately. In cases where passwords or other security-related information has changed, you should also hide any previously displayed information and force the user to enter the new password.

Opting Out of Background Execution

If you do not want your application to remain in the background when it is quit, you can explicitly opt out of the background execution model by adding the `UIApplicationExitsOnSuspend` key to your application's `Info.plist` file and setting its value to `YES`. When an application opts out, it cycles between the not running, inactive, and active states and never enters the background or suspended states. When the user taps the Home button to quit the application, the `applicationWillTerminate:` method of the application delegate is called and the application has approximately five seconds to clean up and exit before it is terminated and moved back to the not running state.

Opting out of background execution is strongly discouraged but may be preferable under certain conditions. Specifically, if coding for the background requires adding significant complexity to your application, terminating the application may be a simpler solution. Also, if your application consumes a large amount of memory and

cannot easily release any of it, the system might need to terminate your application quickly anyway to make room for other applications. Thus, opting to terminate, instead of switching to the background, might yield the same results and save you development time and effort.

Note: Explicitly opting out of background execution is necessary only if your application is linked against iOS SDK 4 and later. Applications linked against earlier versions of the SDK do not support background execution as a rule and therefore do not need to opt out explicitly.

For more information about the keys you can include in your application's `Info.plist` file, see *Information Property List Key Reference*.

Windows, Views, and View Controllers

You use windows and views to present your application's visual content on the screen and to manage the immediate interactions with that content. A **window** is an instance of the `UIWindow` class. All iOS applications have at least one window, and some applications may have additional windows to manage specific types of content. An application's main window fills the entire main screen and has no visual adornments such as a title bar or close box. A window is simply a blank canvas that you use to host one or more views. Therefore, all manipulations to windows must occur through the programmatic interfaces of the `UIWindow` class.

A **view**, an instance of the `UIView` class, defines a rectangular region inside a window. Views are the primary mechanism for interacting with the user in your application. Views have several responsibilities in your application, including:

- Drawing and animation support
 - Views draw content in their rectangular area.
 - Some view properties can be animated to new values.
- Layout and subview management
 - Views manage a list of subviews, allowing you to create arbitrary hierarchies.
 - Views define their own resizing behaviors in relation to their parent view.
 - Views can change the size and position of their subviews automatically or using custom algorithms you define.
- Event handling
 - Views receive touch events.
 - Views forward events to other objects when appropriate.

View controllers play an important part of your application's overall design and structure. Applications running on iOS-based devices have a limited amount of screen space in which to display content. To better manage that screen space, an application uses view controller objects to coordinate the presentation of self-contained view hierarchies. When you want to make significant changes to your application's content,

rather than change the underlying view objects manually, which is tedious and error prone, you simply present the contents of a new view controller. View controller objects are descendants of the `UIViewController` class, which is defined in the UIKit framework.

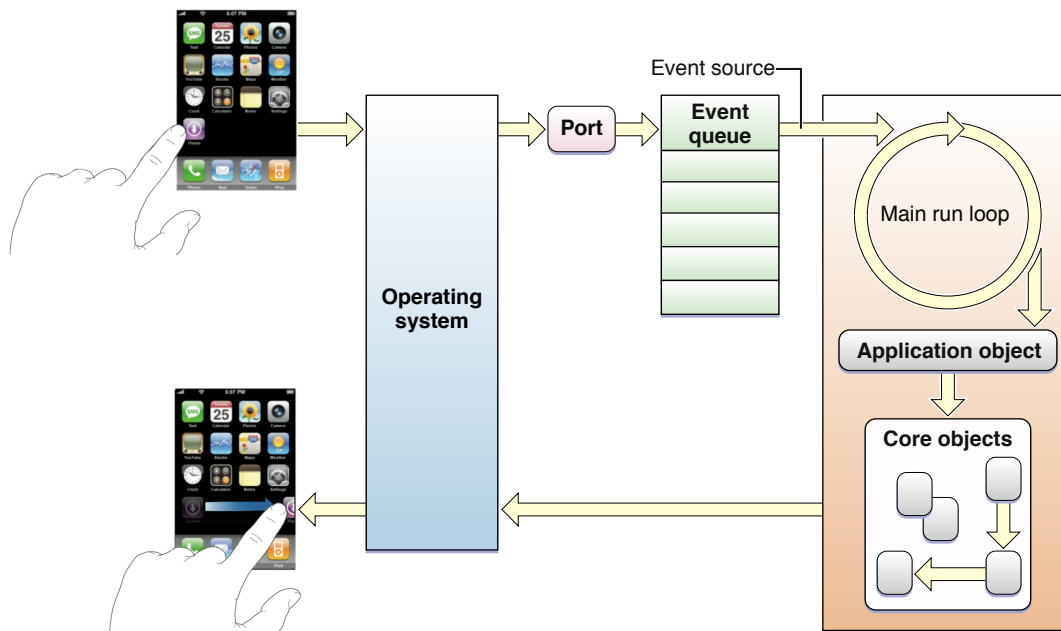
Each view controller object can be thought of as an island unto itself. It manages the presentation of its views, handles their creation and destruction, and facilitates the interactions between those views and other objects in your application. The view controller manages a single top-level view directly and may manage all or some of that view's subviews. For simple user interfaces, a view controller typically manages all of the views in its view hierarchy. However, for more complex interfaces comprised of several distinct pieces, a view controller may manage a subset of views and rely on one or more custom controller objects to manage other groups of views in the view hierarchy.

Understanding view controllers, and the infrastructure they provide, is important to developing iOS applications. For more information about this infrastructure and how you use view controllers to manage your application's user interface, see *View Controller Programming Guide for iOS*. For more information about implementing windows and views, see *View Programming Guide for iOS*.

The Event-Handling System

The iOS event-handling system is responsible for tracking touch and motion events and delivering them to your application. All events are delivered to the application through the `UIApplication` object, which manages the queue of incoming events and distributes them to other parts of the application. For most applications, touches are the most significant type of event you can receive. Other types of events may also be generated and delivered, but touches reflect direct interactions with your application's views.

When it launches an application, the system creates both a process and a single thread for the application. This initial thread becomes the application's main thread. This is where the `UIApplication` object sets up the **main run loop** and configures its event-handling code, as shown in Figure 2-7. As touch events come into the application, they are queued until the application is ready to process them. The application processes events in the main run loop to ensure that they are handled sequentially as they arrive. The actual handling of a given event usually occurs in other objects, such as your application's views and view controllers.

Figure 2-7 Processing events in the main run loop

Note: A run loop monitors sources of input for a given thread of execution. The application's event queue represents one of these input sources. When an event arrives, the run loop wakes up the thread and dispatches control to the handler for the event queue, which in this case is the `UIApplication` object. When the handler finishes, control passes back to the run loop, which then processes another event, processes other input sources, or puts the thread to sleep if there is nothing more to do. For more information about how run loops and input sources work, see *Threading Programming Guide*.

Every event sent to an application is encapsulated in a single event object (`UIEvent`). In the case of touch-related events, the event object contains one or more touch objects (`UITouch`) representing the fingers that are touching the screen. As the user places fingers on the screen, moves them around, and finally removes them from the screen, the system reports the changes for each finger in the corresponding touch object.

Distributing and handling events is the job of responder objects, which are instances of the `UIResponder` class. The `UIApplication`, `UIViewController`, `UIWindow`, and `UIView` classes are all descendants of `UIResponder`. After pulling an event off the event queue, the application dispatches that event to the `UIWindow` object where it occurred. The window object, in turn, forwards the event to its **first responder**. In the case of touch events, the first responder is typically the view object (`UIView`) in which the touch took place. For example, a touch event occurring in a button is delivered to the corresponding button object.

If the first responder is unable to handle an event, it forwards the event to its **next responder**, which is typically a parent view or view controller. If that object is unable to handle the event, it forwards it to its next responder, and so on until the event is handled. This series of linked responder objects is known as the **responder chain**. Messages continue traveling up the responder chain—toward higher-level responder objects such as the window, the application, and the application's delegate—until the event is handled. If the event isn't handled, it is discarded.

The responder object that handles an event tends to set in motion a series of programmatic actions by the application. For example, a control object (that is, a subclass of `UIControl`) handles an event by sending an action message to another object, typically the controller that manages the current set of active views. While

processing the action message, the controller might change the user interface or adjust the position of views in ways that require some of those views to redraw themselves. When this happens, the view and graphics infrastructure takes over and processes the required redraw events in the most efficient manner possible.

For more information about events, responders, and how you handle events in your own custom objects, see *Event Handling Guide for iOS*.

The Graphics and Drawing System

There are two basic drawing paths an application can follow in iOS:

- Use native drawing technologies (such as Core Graphics and UIKit).
- Use OpenGL ES.

The native drawing technologies rely on the infrastructure provided by views and windows to render and present custom content. When a view is first shown, the system asks it to draw its content. System views draw their contents automatically, but for custom views, the system calls the `drawRect:` method of the view. Inside this method, you use the native drawing technologies to draw shapes, text, images, gradients, or any other visual content you want. When you want to change the contents of the view, you do not call the `drawRect:` method yourself. Instead, you ask the system to redraw all or part of the view by calling the `setNeedsDisplay` or `setNeedsDisplayInRect:` method. This cycle then repeats and continues throughout the lifetime of your application.

If you are using OpenGL ES to draw your application's content, you still create a window and view to manage your content but you use them in a different way. Instead of the view drawing your application's content, the view simply provides the rendering surface that you then use to create an OpenGL drawing context. You then use that context and your own timing information to coordinate updates to your application's content.

For information about how to use views and the native rendering technologies to draw your application's content, see *View Programming Guide for iOS*. For detailed information about how to use OpenGL ES to draw your application's content, see *OpenGL ES Programming Guide for iOS*.

The Text System

The text system in iOS provides everything you need to receive input from the user and display the resulting text in your application. On the input side, the system handles text input through the system keyboard, which is tied to the first responder object. Although it is referred to as the *keyboard*, its appearance does not always look like a traditional keyboard. Different languages have different text input requirements, and so the keyboard adapts as needed to support different input methods. Figure 2-8 shows several different variants of the system keyboard that are presented automatically based on the user's current language settings.

Figure 2-8 Several different keyboards and input methods

For displaying text, applications have a variety of options. For simple text display and editing, you can use UIKit classes such as `UILabel`, `UITextField`, and `UITextView`. You can also do simple text display using special additions to the `NSString` class provided by UIKit. For more sophisticated layout, you can use the Core Graphics text facilities or Core Text. Both frameworks provide drawing primitives to render strings of text. In addition, the Core Text framework provides a sophisticated layout engine for computing line positions, text runs, and page layout information that you can then use when drawing the text.

For information about how to support text input and rendering in your application, see *Text and Web Programming Guide for iOS*.

Audio and Video Support

Whether multimedia features are central or incidental to your application, iOS users expect high quality. When presenting video content, take advantage of the device's high-resolution screen and high frame rates. When designing the audio portion of your application, keep in mind that compelling sound adds immeasurably to a user's overall experience.

You can use the iOS multimedia frameworks to add features like:

- High-quality audio recording, playback, and streaming
- Immersive game sounds
- Live voice chat
- Playback of content from a user's iPod library
- Full-screen video playback
- Video recording on devices that support it

For more information about the audio and video technologies available in iOS, see *iOS Technology Overview*. For information about how to use the audio and video technologies in iOS, see *Multimedia Programming Guide*.

Integration with the Hardware and System Applications

An iOS application does not need to be isolated from the rest of the system. In fact, the best applications take advantage of both hardware and software features on the device to provide a more intimate experience for the user. Devices contain a lot of user-specific information, much of which is accessible through the system frameworks and technologies. Table 2-5 lists a few of the features you can incorporate into your applications.

Table 2-5 System integration technologies

Integration with...	Description
The user's contacts	Applications that need access to the user's contacts can use the Address Book framework to access that information. You can also use the Address Book UI framework to present standard system interfaces for picking and creating contacts. For more information, see <i>Address Book Programming Guide for iOS</i> .
Systemwide calendar and time-based events	Applications that need to schedule time-based events can use the Event Kit and Event Kit UI frameworks to do so. Events scheduled using this framework show up in the Calendar application and other applications that support this framework.
The Mail and Messages applications	If your application sends email or SMS messages, you can use the view controllers in the Message UI framework to present the standard system interfaces for composing and sending those messages.

Integration with...	Description
Telephony information	Applications that need information about the telephony features of a device can access that information using the Core Telephony framework. For example, a VoIP application might use this capability to detect an in-progress cellular call and handle VoIP calls differently.
The camera hardware and the user's photo library	Applications that need access to the camera or the user's photo library can access them both using the <code>UIImagePickerController</code> class. This class presents a standard system interface for retrieving images from the user.
Unknown file types	If your application interacts with unknown file types, you can use the <code>UIDocumentInteractionController</code> class to preview the contents of the files or find an application capable of opening them. Email applications and other network-based applications are typical candidates for interacting with unknown file types.
Pasteboard data	The pasteboard is a way of moving information around inside an application but is also a way to share information with other applications. Data on the pasteboard can be copied into other applications and incorporated.
The location of the device	Applications can take advantage of location-based data to tailor the content presented to the user. For example, searches for local restaurants or services can be limited to nearby places only. Location services are also used frequently to create social connections by showing the location of nearby users. For information about using Core Location, see <i>Location Awareness Programming Guide</i> .
Map information	Applications can incorporate maps into their applications and layer information on top of those maps using the Map Kit framework. For information about using Map Kit, see <i>Location Awareness Programming Guide</i> .
External hardware accessories	Developers can create software that interacts with connected external hardware using the External Accessory framework. For information about communicating with external accessories, see <i>External Accessory Programming Topics</i> .

For a complete list of technologies you can incorporate into your applications, see *iOS Technology Overview*.

Implementing Standard Application Behaviors

There are two standard behaviors that you should always implement for an iOS application and there are others that you may want to implement depending on your needs. The two you should always implement are:

- Observe low-memory warnings and release as much memory as possible.
- Save your application's state information before moving to the background.

This chapter provides examples of these and other common application behaviors and offers guidance on how best to implement those behaviors. This chapter also demonstrates the proper way to implement common tasks that all applications typically need to perform.

Observing Low-Memory Warnings

When the system dispatches a low-memory warning to your application, heed it. iOS notifies the frontmost application whenever the amount of free memory dips below a safe threshold. If your application receives this warning, it must free up as much memory as it can by releasing objects it does not need or clearing out memory caches that it can easily recreate later.

UIKit provides several ways to receive low-memory warnings, including the following:

- Implement the `applicationDidReceiveMemoryWarning:` method of your application delegate.
- Override the `didReceiveMemoryWarning` method in your custom `UIViewController` subclass.
- Register to receive the `UIApplicationDidReceiveMemoryWarningNotification` notification.

Upon receiving any of these warnings, your handler method should respond by immediately freeing up any unneeded memory. For example, the `UIViewController` class responds by automatically purging its view if that view is not currently visible; subclasses can supplement the default behavior by purging additional data structures. An application that maintains a cache of images might respond by releasing any images that are not currently onscreen.

If your custom objects have known purgeable resources, you can have those objects register for the `UIApplicationDidReceiveMemoryWarningNotification` notification and release their purgeable resources directly. Have these objects register if you have a few objects that manage most of your purgeable resources and if it is appropriate to purge all of those resources. But if you have many purgeable objects or want to coordinate the release of only a subset of those objects, you might want to use your application delegate to release the desired objects.

Important: Like the system applications, your applications should always handle low-memory warnings. System applications consume small amounts of memory while processing requests. When a low-memory condition is detected, the system delivers low-memory warnings to all running programs (including your application) and may terminate some background applications (if necessary) to ease memory pressure. If not enough memory is released—perhaps because your application is leaking or still consuming too much memory—the system may still terminate your application.

You can test your application's behavior under low-memory conditions using the Simulate Memory Warning command in the simulator.

Preserving the Application's Current State

Whenever your application moves to the background, you need to save to disk any data that might be lost if something were to happen to your application. Background applications can be purged by the system at any time if memory pressure becomes strong enough. Suspended applications are usually purged first but other applications can be purged too, especially if they consume large amounts of memory themselves. Writing out any unsaved user data or application state information ensures that your application has enough data to restore itself to current state upon a subsequent relaunch.

In addition to writing out data when moving to the background, your application should also save changes to user data at key points in your workflow. Unlike most desktop applications, there is no save command in an iOS application. Therefore changes made by the user need to be written out to disk at the point where the user makes or confirms the change. Under no circumstances should you let the user navigate to a new page of content without saving data on the current page first.

Saving the state of your application's user interface and data structures is also something you should do to improve the user experience. Because iOS applications are meant to be launched and used quickly, restoring your application to the state it was in when it was last run provides a continuity between launches. You do not necessarily have to return the user to the exact same screen as before, but you should return the user to the most logical starting point. For example, if a user edits a contact and then leaves the Phone application, upon returning, the Phone application displays the top-level list of contacts, rather than the editing screen for the contact.

Launching in Landscape Mode

Applications in iOS normally launch in portrait mode to match the orientation of the Home screen. If you have an application that runs in both portrait and landscape mode, your application should always launch in portrait mode initially and then let its view controllers rotate the interface as needed based on the device's orientation. If your application runs in landscape mode only, however, you must perform the following steps to make it launch in a landscape orientation initially:

- In your application's `Info.plist` file, add the `UIInterfaceOrientation` key and set its value to the landscape mode. For landscape orientations, you can set the value of this key to `UIInterfaceOrientationLandscapeLeft` or `UIInterfaceOrientationLandscapeRight`.
- Lay out your views in landscape mode and make sure that their autosizing options are set correctly.
- Override your view controller's `shouldAutorotateToInterfaceOrientation:` method and return `YES` only for the desired landscape orientation and `NO` for portrait orientations.

Important: The preceding steps assume your application uses view controllers to manage its view hierarchy. View controllers provide a significant amount of infrastructure for handling orientation changes as well as other complex view-related events. If your application is not using view controllers—as may be the case with games and other OpenGL ES–based applications—you are responsible for rotating the drawing surface (or adjusting your drawing commands) as needed to present your content in landscape mode.

The `UIInterfaceOrientation` property hints to iOS that it should configure the orientation of the application status bar (if one is displayed) as well as the orientation of views managed by any view controllers at launch time. In iOS 2.1 and later, view controllers respect this property and set their view's initial orientation to match. Using this property is also equivalent to calling the `setStatusBarOrientation:animated:` method of `UIApplication` early in the execution of your `applicationDidFinishLaunching:` method.

Note: To launch a view controller–based application in landscape mode in versions of iOS prior to v2.1, you need to apply a 90-degree rotation to the transform of the application's root view in addition to all the preceding steps. Prior to iOS 2.1, view controllers did not automatically rotate their views based on the value of the `UIInterfaceOrientation` key.

Files and the File System

Every application has a protected area in which it can create and modify files. In addition, the system allows applications to share files with each other (and with the user) through well-defined and secure means. However, supporting those means requires work on your part. The following sections describe the types of file-related operations you can perform in your applications.

Getting Paths to Application Directories

At various levels of the system, there are programmatic ways to obtain file-system paths to the directory locations of an application's sandbox. However, the preferred way to retrieve these paths is with the Cocoa programming interfaces.

You can use the `NSSearchPathForDirectoriesInDomains` function to get exact paths to your `Documents` and `Caches` directories. To get a path to the `tmp` directory, use the `NSTemporaryDirectory` function. When you have one of these “base” paths, you can use the path-related methods of `NSString` to modify the path information or create new path strings. For example, upon retrieving the temporary directory path, you could append a file name and use the resulting string to create a file with the given name in the temporary directory.

Note: If you are using frameworks with ANSI C programmatic interfaces—including those that take paths—recall that `NSString` objects are “toll-free bridged” with their Core Foundation counterparts. This means that you can cast a `NSString` object (such as the return by one of the above functions) to a `CFStringRef` type, as shown in the following example:

```
CFStringRef tmpDirectory = (CFStringRef)NSTemporaryDirectory();
```

The `NSSearchPathForDirectoriesInDomains` function of the Foundation framework lets you obtain the full path to several application-related directories. To use this function in iOS, specify an appropriate search path constant for the first parameter and `NSUserDomainMask` for the second parameter. Table 3-1 lists several of the most commonly used constants and the directories they return.

Table 3-1 Commonly used search path constants

Constant	Directory
<code>NSDocumentDirectory</code>	<code><Application_Home>/Documents</code>
<code>NSCachesDirectory</code>	<code><Application_Home>/Library/Caches</code>
<code>NSApplicationSupportDirectory</code>	<code><Application_Home>/Library/Application Support</code>

Because the `NSSearchPathForDirectoriesInDomains` function was designed originally for Mac OS X, where multiple such directories could exist, it returns an array of paths rather than a single path. In iOS, the resulting array should contain the single path to the desired directory. Listing 3-1 shows a typical use of this function.

Listing 3-1 Getting the path to the application's Documents directory

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

You can call `NSSearchPathForDirectoriesInDomains` using a domain-mask parameter other than `NSUserDomainMask` or a directory constant other than those in Table 3-1 (page 50), but the application will be unable to write to any of the returned directory locations. For example, if you specify `NSApplicationDirectory` as the directory parameter and `NSSystemDomainMask` as the domain-mask parameter, you get the path `/Applications` returned (on the device), but your application cannot write any files to this location.

Another consideration to keep in mind is the difference in directory locations between platforms. The paths returned by the `NSSearchPathForDirectoriesInDomains` function differ depending on whether you're running your application on the device or on the simulator. For example, take the function call shown in Listing 3-1 (page 50). On the device, the returned path (`documentsDirectory`) is similar to the following:

```
/var/mobile/Applications/30B51836-D2DD-43AA-BCB4-9D4DADFED6A2/Documents
```

However, on the Simulator, the returned path takes the following form:

```
/Volumes/Stuff/Users/johnDoe/Library/Application Support/iPhone
Simulator/User/Applications/118086A0-FAAF-4CD4-9A0F-CD5E8D287270/Documents
```

To read and write user preferences, use the `NSUserDefaults` class or the `CFPreferences` API. These interfaces eliminate the need for you to construct a path to the `Library/Preferences/` directory and read and write preference files directly. For more information on using these interfaces, see “Adding the Settings Bundle” (page 86).

If your application contains sound, image, or other resources in the application bundle, you should use the `NSBundle` class or `CFBundleRef` opaque type to load those resources. Bundles have an inherent knowledge of where resources live inside the application. In addition, bundles are aware of the user's language preferences and are able to choose localized resources over default resources automatically. For more information on bundles, see “The Application Bundle” (page 93).

Sharing Files with the User

Applications that want to make user data files accessible can do so using application file sharing. File sharing enables the application to expose the contents of its `/Documents` directory to the user through iTunes. The user can then move files back and forth between the device and a desktop computer. This feature does not allow your application to share files with other applications on the same device, though. To share data and files between applications, you must use the pasteboard or a document interaction controller object.

To enable file sharing for your application, do the following:

1. Add the `UIFileSharingEnabled` key to your application's `Info.plist` file and set the value of the key to `YES`.
2. Put whatever files you want to share in your application's `Documents` directory.
3. When the device is plugged into the user's computer, iTunes 9.1 displays a File Sharing section in the Apps tab of the selected device.
4. The user can add files to this directory or move files to the desktop.

Applications that support file sharing should be able to recognize when files have been added to the `Documents` directory and respond appropriately. For example, your application might make the contents of any new files available from its interface. You should never present the user with the list of files in this directory and ask them to decide what to do with those files.

Important: In iTunes, users can manipulate only the top-level items in the `Documents` directory. Users cannot see the contents of subdirectories but can delete those directories or copy their contents to the desktop. You should take this into consideration when deciding where to put files in this directory.

Working with Protected Files

When file protection is enabled on a device, applications can mark files as protected for greater security. If you plan to support this feature in your applications, though, you must also be prepare your application to handle situations where the protected file is unavailable. The following sections show you how to add protection to files and how to modify your application to work with protected files.

Marking a File as Protected

To mark a file as protected, you must add an extended attribute to it. The Foundation framework includes two ways to add this attribute:

- When writing the contents of an `NSData` object to disk using the `writeToFile:options:error:` method, include the `NSDataWritingFileProtectionComplete` option.
- Use the `setAttributes:ofItemAtPath:error:` method of `NSFileManager` to add the `NSFileProtectionKey` attribute (with the `NSFileProtectionComplete` value) to an existing file.

Note: If you are working with `NSData` objects already, using the `writeToFile:options:error:` method to save and protect that data in one step is recommended. This guarantees that the data is always stored on disk in an encrypted format.

Adding protection to a file is a one-way operation—once added, you cannot remove the attribute. When adding this attribute to new files, it is recommended that you add this attribute after you create the file but before you write any data to it. (If you are writing out the contents of an `NSData` object, this happens automatically.) When adding this attribute to existing (unprotected) files, adding this key replaces the unprotected file with a protected version.

Determining the Availability of Protected Files

A protected file can be accessed only when a device is unlocked. Because applications may continue running while a device is locked, your code should be prepared to handle the possibility of protected files becoming unavailable at any time. The UIKit framework provides ways to track whether data protection is currently enabled.

- The application delegate can implement the `applicationProtectedDataWillBecomeUnavailable:` and `applicationProtectedDataDidBecomeAvailable:` methods and use them to track changes to the availability of protected data.
- An application can register for the `UIApplicationProtectedDataWillBecomeUnavailable` and `UIApplicationProtectedDataDidBecomeAvailable` notifications.
- The `protectedDataAvailable` property of the shared `UIApplication` object indicates whether protected files are currently accessible.

Any application that works with protected files should implement the application delegate methods. When the `applicationProtectedDataWillBecomeUnavailable:` method is called, your application should immediately close any protected files and refrain from using them again until the `applicationProtectedDataDidBecomeAvailable:` method is called. Any attempts to access the protected files while they are unavailable will fail.

Opening Files Whose Type is Unknown

When your application needs to interact with files of unknown types, you can use a `UIDocumentInteractionController` object to manage those interactions. A document interaction controller works with the system to determine whether files can be previewed in place or opened by another application. Your application works with the document interaction controller to present the available options to the user at appropriate times.

To use a document interaction controller in your application, do the following:

1. Create an instance of the `UIDocumentInteractionController` class for each file you want to manage.
2. Present the file in your application's user interface. (Typically, you would do this by displaying the file name or icon somewhere in your interface.)
3. When the user interacts with the file, ask the document interaction controller to present one of the following interfaces:

- A file preview view that displays the contents of the file
- A menu containing options to preview the file, copy its contents, or open it using another application
- A menu prompting the user to open it with another application

Any application that interacts with files can use a document interaction controller. Programs that download files from the network are the most likely candidates to need these capabilities. For example, an email program might use document interaction controllers to preview or open files attached to an email. Of course, you do not need to download files from the network to use this feature.

Creating and Configuring a Document Interaction Controller

To create a new document interaction controller, initialize a new instance of the `UIDocumentInteractionController` class with the file you want it to manage and assign an appropriate delegate object. Your delegate object is responsible for providing the document interaction controller with information it needs to present its views. You can also use the delegate to perform additional actions when those views are displayed. The following code creates a new document interaction controller and sets the delegate to the current object. Note that the caller of this method needs to retain the returned object.

```
- (UIDocumentInteractionController*)docControllerForFile:(NSURL*)fileURL
{
    UIDocumentInteractionController* docController =
        [UIDocumentInteractionController interactionControllerWithURL:fileURL];
    docController.delegate = self;

    return docController;
}
```

Once you have a document interaction controller object, you can use its properties to get information about the file, including its name, type information, and path information. The controller also has an `icons` property that contains `UIImage` objects representing the document's icon in various sizes. You can use all of this information when presenting the document in your user interface.

If you plan to let the user open the file in another application, you can use the `annotation` property of the document interaction controller to pass custom information to the opening application. It is up to you to provide information in a format that the other application will recognize. For example, this property is typically used by application suites that want to communicate additional information about a file to other applications in the suite. The opening application sees the annotation data in the `UIApplicationLaunchOptionsAnnotationKey` key of the options dictionary that is passed to it at launch time.

Presenting a Document Interaction Controller

When the user interacts with a file, you use the document interaction controller to display the appropriate user interface. You have the choice of displaying a document preview or of prompting the user to choose an appropriate action for the file using one of the following methods:

- Use the `presentOptionsMenuFromRect:inView:animated:` or `presentOptionsMenuFromBarButtonItem:animated:` method to present the user with a variety of options.
- Use the `presentPreviewAnimated:` method to display a document preview.

- Use the `presentOpenInMenuFromRect:inView:animated:` or `presentOpenInMenuFromBarButtonItem:animated:` method to present the user with a list of applications with which to open the file.

Each of the preceding methods attempts to display a custom view with the appropriate content. When calling these methods, you should always check the return value to see if the attempt was actually successful. These methods may return `NO` if the resulting interface would have contained no content. For example, the `presentOpenInMenuFromRect:inView:animated:` method returns `NO` if there are no applications capable of opening the file.

If you choose a method that might display a preview of the file, your delegate object must implement the `documentInteractionControllerViewViewControllerForPreview:` method. Most document previews are displayed using a modal view, so the view controller you return becomes the parent of the modal document preview. (If you return a navigation controller, the document interaction controller is pushed onto its navigation stack instead.) If you do not implement this method, if your implementation returns `nil`, or if the specified view controller is unable to present the document interaction controller, a document preview is not displayed.

Normally, the document interaction controller automatically handles the dismissal of the view it presents. However, you can dismiss the view programmatically as needed by calling the `dismissMenuAnimated:` or `dismissPreviewAnimated:` methods.

Implementing Support for Custom File Formats

Applications that are able to open specific document or file formats may register those formats with the system. When the system or another application needs to open a file, it can hand that file off to your application to do so. In order to support custom file formats, your application must:

1. Register the file types your application supports with the system.
2. Implement the proper methods to open files (when asked to do so by the system).

Registering the File Types Your Application Supports

If your application is capable of opening specific types of files, you should register those types with the system. To declare support for file types, your application must include the `CFBundleDocumentTypes` key in its `Info.plist` file. The system gathers this information from your application and maintains a registry that other applications can access through a document interaction controller.

The `CFBundleDocumentTypes` key contains an array of dictionaries, each of which identifies information about a specific document type. A document type usually has a one-to-one correspondence with a particular document type. However, if your application treats more than one file type the same way, you can group those types together as a single document type. For example, if you have two different file formats for your application's native document type, you could group both the old type and new type together in a single document type entry. By doing so, both the new and old files would appear to be the same type of file and would be treated in the same way.

Each dictionary in the `CFBundleDocumentTypes` array can include the following keys:

- `CFBundleTypeName` specifies the name of the document type.

- `CFBundleTypeIconFiles` is an array of filenames for the image resources to use as the document's icon. For information on how to create document icons for your application, see *iPhone Human Interface Guidelines* and *iPad Human Interface Guidelines*.
- `LSItemContentTypes` contains an array of strings with the UTI types that represent the supported file types in this group.
- `LSHandlerRank` describes whether this application owns the document type or is merely able to open it.

From the perspective of your application, a document is a file type (or file types) that the application supports and treats as a single entity. For example, an image processing application might treat different image file formats as different document types so that it can fine tune the behavior associated with each one. Conversely, a word processing application might not care about the underlying image formats and just manage all image formats using a single document type.

Table 6-2 shows a sample XML snippet from the `Info.plist` of an application that is capable of opening a custom file type. The `LSItemContentTypes` key identifies the UTI associated with the file format and the `CFBundleTypeIconFiles` key points to the icon resources to use when displaying it.

Listing 3-2 Document type information for a custom file format

```
<dict>
  <key>CFBundleTypeName</key>
  <string>My File Format</string>
  <key>CFBundleTypeIconFiles</key>
  <array>
    <string>MySmallIcon.png</string>
    <string>MyLargeIcon.png</string>
  </array>
  <key>LSItemContentTypes</key>
  <array>
    <string>com.example.myformat</string>
  </array>
  <key>LSHandlerRank</key>
  <string>Owner</string>
</dict>
```

For more information about the contents of the `CFBundleDocumentTypes` key, see the description of that key in *Information Property List Key Reference*.

Opening Supported File Types

The system may ask your application to open a specific file and present it to the user at any time. Requests to open another document might cause your application to be launched or, if your application is already running in the background, cause it to move to the foreground. In both cases, the system provides information about the file to be opened to your application delegate.

There are two application delegate methods to implement for handling the opening of a file:

- Use the `application:didFinishLaunchingWithOptions:` method to retrieve information about the file and decide whether you want to open it.
- Use the `application:handleOpenURL:` method to open the file.

The `application:didFinishLaunchingWithOptions:` method is called only when your application needs to be launched to open a file. Your implementation of this method should retrieve any relevant keys from the options dictionary and use them to decide if it should open the file. You typically do not open the file in this method, though. Instead, you return YES to indicate that your application is able to open the file. If you return YES, the system calls your `application:handleOpenURL:` method to open the file. However, if you return NO, the `application:handleOpenURL:` method is not called.

The options dictionary for the `application:didFinishLaunchingWithOptions:` method may contain several keys with information about the file and the application that asked for it to be opened. Specifically, this dictionary may contain the following keys:

- The value for the `UIApplicationLaunchOptionsURLKey` key is an `NSURL` object that specifies the file to open.
- The value for the `UIApplicationLaunchOptionsSourceApplicationKey` key is an `NSString` with the bundle identifier of the application that initiated the open request.
- The value for the `UIApplicationLaunchOptionsAnnotationKey` is a property list object that the source application associated with the file. This object contains relevant information the application wanted to communicate to other applications.

Modifying Documents Passed to Your Application by the System

When you use a `UIDocumentInteractionController` to open a file, the system transfers the file to the `~/Documents/Inbox` directory of the application asked to open it. Because the `Inbox` directory is managed by the system, only the system is allowed to write to it. The application opening the file has read-only access to the file and cannot write to it.

If your application allows the user to edit documents passed to it by a document interaction controller, you must copy those documents to a writable directory before attempting to save any changes. Do not copy a file out of the `Inbox` directory until the user actually makes a change to it. And the process of copying files out of the `Inbox` directory (and to a writable directory) should be transparent to the user. You should not prompt the user about whether to save the file to a different location.

Communicating with Other Applications

If an application handles URLs of a known type, you can use that URL scheme to communicate with the application. In most cases, however, URLs are used simply to launch another application and have it display some information that is relevant to your own application. For example, if your application manages address information, you could send a URL containing a given address to the Maps application to show that location. This level of communication creates a much more integrated environment for the user and alleviates the need for your application to implement features that exist elsewhere on the device.

Apple provides built-in support for the `http`, `mailto`, `tel`, and `sms` URL schemes. It also supports `http`-based URLs targeted at the Maps, YouTube, and iPod applications. Applications can register their own custom URL schemes as well. To communicate with an application, create an `NSURL` object with some properly formatted content and pass it to the `openURL:` method of the shared `UIApplication` object. The `openURL:` method launches the application that has registered to receive URLs of that type and passes it the URL. When the user subsequently quits that application, the system often relaunches your application but may not always do so. The decision to relaunch an application is made based on the user's actions in the handler application and whether returning to your application would make sense from the user's perspective.

The following code fragment illustrates how one application can request the services of another application (“todolist” in this example is a hypothetical custom scheme registered by an application):

```
NSURL *myURL = [NSURL
URLWithString:@"todolist://www.acme.com?Quarterly%20Report#200806231300"];
[[UIApplication sharedApplication] openURL:myURL];
```

Important: If your URL type includes a scheme that is identical to one defined by Apple, the Apple-provided application is launched instead of your application. If multiple third-party applications register to handle the same URL scheme, it is undefined as to which of the applications is picked to handle URLs of that type.

If your application defines its own custom URL scheme, it should implement a handler for that scheme as described in “[Implementing Custom URL Schemes](#)” (page 57). For more information about the system-supported URL schemes, including information about how to format the URLs, see *Apple URL Scheme Reference*.

Implementing Custom URL Schemes

You can register URL types for your application that include custom URL schemes. A custom URL scheme is a mechanism through which third-party applications can interact with each other and with the system. Through a custom URL scheme, an application can make its services available to other applications.

Registering Custom URL Schemes

To register a URL type for your application, you must specify the subproperties of the `CFBundleURLTypes` property, which was introduced in “[The Information Property List](#)” (page 95). The `CFBundleURLTypes` property is an array of dictionaries in the application’s `Info.plist` file, with each dictionary defining a URL type the application supports. Table 3-2 describes the keys and values of a `CFBundleURLTypes` dictionary.

Table 3-2 Keys and values of the `CFBundleURLTypes` property

Key	Value
<code>CFBundleURLName</code>	<p>A string that is the abstract name for the URL type. To ensure uniqueness, it is recommended that you specify a reverse-DNS style of identifier, for example, <code>com.acme.myscheme</code>.</p> <p>The URL-type name provided here is used as a key to a localized string in the <code>InfoPlist.strings</code> file in a language-localized bundle subdirectory. The localized string is the human-readable name of the URL type in a given language.</p>
<code>CFBundleURLSchemes</code>	An array of URL schemes for URLs belonging to this URL type. Each scheme is a string. URLs belonging to a given URL type are characterized by their scheme components.

Figure 3-1 shows the `Info.plist` file of an application being edited using the built-in Xcode editor. In this figure, the URL types entry in the left column is equivalent to the `CFBundleURLTypes` key you would add directly to an `Info.plist` file. Similarly, the “URL identifier” and “URL Schemes” entries are equivalent to the `CFBundleURLName` and `CFBundleURLSchemes` keys.

Figure 3-1 Defining a custom URL scheme in the `Info.plist` file

Key	Value
▼ Information Property List	(12 items)
Localization native develo	en
Bundle display name	\$(PRODUCT_NAME)
Executable file	\$(EXECUTABLE_NAME)
Icon file	
Bundle identifier	com.acme.\$(PRODUCT_NAME)
InfoDictionary version	6.0
Bundle name	\$(PRODUCT_NAME)
Bundle OS Type code	APPL
Bundle creator OS Type co	????
Bundle version	1.0
Main nib file base name	MainWindow
▼ URL types	(1 item)
▼ Item 1	(2 items)
URL identifier	com.acme.ToDoList
▼ URL Schemes	(1 item)
Item 1	todolist

After you have registered a URL type with a custom scheme by defining the `CFBundleURLTypes` property, you can test the scheme in the following way:

1. Build, install, and run your application.
2. Go to the Home screen and launch Safari. (In the simulator, you can go to the Home screen by selecting Hardware > Home from the menu.)
3. In the address bar of Safari, type a URL that uses your custom scheme.
4. Verify that your application launches and that the application delegate is sent a `application:handleOpenURL: message`.

Handling URL Requests

An application that has its own custom URL scheme must be able to handle URLs passed to it. All URLs are passed to your application delegate for handling and the URL can be passed in when your application is launched or while it is running in the background. Your delegate should implement the following methods to handle incoming URLs:

- Use the `application:didFinishLaunchingWithOptions:` method to retrieve information about the URL and decide whether you want to open it. This method is called only when your application is launched.
- Use the `application:handleOpenURL:` method to open the URL.

If your application is not running when a URL request arrives, it is launched and moved to the foreground so that it can open the URL. If your application is running but is either in the background or suspended, it is similarly moved to the foreground to handle the URL. In both cases, you use the `application:handleOpenURL:` to handle the actual opening of the URL. In the case where your application is launched, the `application:didFinishLaunchingWithOptions:` method is also called and receives the URL in its options dictionary but it should use this information only to determine whether your application is able to open the URL.

Note: When your application launches because of a request to open a URL, you can display a custom launch image for each scheme you define. For more information about how to specify these launch images, see [“Providing Launch Images for Custom URL Schemes”](#) (page 104).

All URLs are passed to your application in an `NSURL` object. It is up to you to define the format of the URL contents, but the `NSURL` class conforms to the RFC 1808 specification and therefore supports most URL formatting conventions. Specifically, it includes methods that return the various parts of a URL as defined by RFC 1808, including the user, password, query, fragment, and parameter strings. The “protocol” for your custom scheme can use these URL parts for conveying various kinds of information.

In the implementation of `application:handleOpenURL:` shown in Listing 3-3, the passed-in URL object conveys application-specific information in its query and fragment parts. The delegate extracts this information—in this case, the name of a to-do task and the date the task is due—and with it creates a model object of the application.

Listing 3-3 Handling a URL request based on a custom scheme

```
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url {
    if ([[url scheme] isEqualToString:@"todolist"]) {
        ToDoItem *item = [[ToDoItem alloc] init];
        NSString *taskName = [url query];
        if (!taskName || ![self isValidTaskString:taskName]) { // must have a
task name
            [item release];
            return NO;
        }
        taskName = [taskName
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

        item.todoTask = taskName;
        NSString *dateString = [url fragment];
        if (!dateString || [dateString isEqualToString:@"today"]) {
            item.dateDue = [NSDate date];
        } else {
            if (![self isValidDateString:dateString]) {
                [item release];
                return NO;
            }
            // format: yyyyymmddhhmm (24-hour clock)
            NSString *curStr = [dateString substringWithRange:NSMakeRange(0,
4)];

            NSInteger yeardigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(4, 2)];
            NSInteger monthdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(6, 2)];
            NSInteger daydigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(8, 2)];
            NSInteger hourdigit = [curStr integerValue];
            curStr = [dateString substringWithRange:NSMakeRange(10, 2)];
            NSInteger minutedigit = [curStr integerValue];

            NSDateComponents *dateComps = [[NSDateComponents alloc] init];
            [dateComps setYear:yeardigit];
            [dateComps setMonth:monthdigit];
            [dateComps setDay:daydigit];
```

```

        [dateComps setHour:hourdigit];
        [dateComps setMinute:minutedigit];
        NSCalendar *calendar = [NSCalendar currentCalendar];
        NSDate *itemDate = [calendar dateFromComponents:dateComps];
        if (!itemDate) {
            [dateComps release];
            [item release];
            return NO;
        }
        item.dateDue = itemDate;
        [dateComps release];
    }

    [(NSMutableArray *)self.list addObject:item];
    [item release];
    return YES;
}
return NO;
}

```

Be sure to validate the input you get from URLs passed to your application; see “Validating Input” in *Secure Coding Guide* to find out how to avoid problems related to URL handling. To learn about URL schemes defined by Apple, see *Apple URL Scheme Reference*.

Displaying Application Preferences

If your application uses preferences to control various aspects of its behavior, how you expose those preferences to the user depends on how integral they are to your program.

- Preferences that are integral to using the application (and simple enough to implement directly) should be presented directly by your application using a custom interface.
- Preferences that are not integral, or that require a relatively complex interface, should be presented using the system’s Settings application.

When determining whether a set of preferences is integral, think about the intended usage pattern. If you expect the user to make changes to preferences somewhat frequently, or if those preferences play a relatively important role in how the application behaves, they are probably integral. For example, the settings in a game are usually integral to playing the game and something the user might want to change quickly. Because the Settings application is a separate application, however, you would use it only for preferences that you do not expect the user to access frequently.

If you choose to implement preferences inside your application, it is up to you to define the interface and write the code to manage those preferences. If you choose to use the Settings application, however, your application must provide a Settings bundle to manage them.

A **settings bundle** is a custom resource that you include in the top level of your application’s bundle directory. An opaque directory with the name `Settings.bundle`, the settings bundle contains specially formatted data files (and supporting resources) that tell the Settings application how to display your preferences. These files also tell the Settings application where to store the resulting values in the preferences database, which your application then accesses using the `NSUserDefaults` or `CFPreferences` APIs.

If you implement your preferences using a settings bundle, you should also provide a custom icon for your preferences. The Settings application displays the image you provide next to your application name. For information about application icons and how you specify them, see [“Application Icons”](#) (page 100).

For more information about creating a Settings bundle for your application, see [“Implementing Application Preferences”](#) (page 81).

Turning Off Screen Locking

If an iOS-based device does not receive touch events for a specified period of time, it turns off the screen and disables the touch sensor. Locking the screen in this way is an important way to save power. As a result, you should leave this feature enabled except when absolutely necessary to prevent unintended behavior in your application. For example, you might disable screen locking if your application does not receive screen events regularly but instead uses other features (such as the accelerometers) for input.

To disable screen locking, set the `idleTimerDisabled` property of the shared `UIApplication` object to `YES`. Be sure to reset this property to `NO` at times when your application does not need to prevent screen locking. For example, you might disable screen locking while the user is playing a game, but you should reenable it when the user is in setup screens or is otherwise not actively playing the game.

Executing Code in the Background

Most applications that enter the background state are immediately suspended and do not execute any code. If your application does need to execute code, it must notify the system of that fact. There are several options available for running background tasks and each option allows your application to support specific types of behavior.

Important: If you need only minimal support for running under multitasking, and do not actually need to perform tasks while in the background, you do not need to read this chapter. Information about how to design your application to support the multitasking environment is available in [“Multitasking”](#) (page 36).

Preparing Your Application to Execute in the Background

Applications linked against iOS 4 and later are automatically assumed to support multitasking and to implement the appropriate methods to handle transitions to the background state. However, work is still required to ensure that your application transitions smoothly between the foreground and background.

Determining Whether Multitasking Support is Available

The ability to run background tasks is not supported on all iOS-based devices. If a device is not running iOS 4 and later, or if the hardware is not capable of running applications in the background, the system reverts to the previously defined behavior for handling applications. Specifically, when an application is quit, its application delegate receives an `applicationWillTerminate:` message, after which the application is terminated and purged from memory.

Applications should be prepared to handle situations where multitasking is not available. If your code supports background tasks but is able to run without them, you can use the `multitaskingSupported` property of the `UIDevice` class to determine whether multitasking is available. Of course, if your application supports versions of the system earlier than iOS 4, you should always check the availability of this property before accessing it, as shown in Listing 4-1.

Listing 4-1 Checking for background support on earlier versions of iOS

```

UIDevice* device = [UIDevice currentDevice];
BOOL backgroundSupported = NO;
if ([device respondsToSelector:@selector(isMultitaskingSupported)])
    backgroundSupported = device.multitaskingSupported;

```

Declaring the Background Tasks You Support

Support for some types of background execution must be declared in advance by the application that uses them. An application does this by including the `UIBackgroundModes` key in its `Info.plist` file. This key identifies which background tasks your application supports. Its value is an array that contains one or more strings with the following values:

- `audio` - The application plays audible content to the user while in the background.
- `location` - The application keeps users informed of their location, even while running in the background.
- `voip` - The application provides the ability for the user to make phone calls using an Internet connection.

Each of the preceding values lets the system know that your application should be woken up at appropriate times to respond to relevant events. For example, an application that begins playing music and then moves to the background still needs execution time to fill the audio output buffers. Including the `audio` key tells the system frameworks that they should continue playing and make the necessary callbacks to the application at appropriate intervals. If the application did not include this key, any audio being played by the application would stop when the application moved to the background.

In addition to the preceding keys, iOS provides two other ways to do work in the background:

- Applications can ask the system for extra time to complete a given task.
- Applications can schedule local notifications to be delivered at a predetermined time.

For more information about how to initiate background tasks from your code, see [“Initiating Background Tasks”](#) (page 66).

Supporting Background State Transitions

Supporting the background state transition is part of the fundamental architecture for applications in iOS 4 and later. Although technically the only thing you have to do to support this capability is link against iOS 4 and later, properly supporting it requires some additional work. Specifically, your application delegate should implement the following methods and implement appropriate behaviors in each of them:

- `application:didFinishLaunchingWithOptions:`
- `applicationDidBecomeActive:`
- `applicationWillResignActive:`
- `applicationDidEnterBackground:`
- `applicationWillEnterForeground:`
- `applicationWillTerminate:`

These methods are called at key times during your application’s execution to let you know when your application is changing states. Although many of these methods have been present in iOS applications for some time, their purpose was more narrowly defined previously. Applications implementing these methods now must use them to address some additional behaviors that are needed to operate safely in the background. For example, an application that went into the background and purged from memory might be relaunched and expected to continue operating from where it last was. Thus, your application’s

`application:didFinishLaunchingWithOptions:` method may now need to check and see what additional launch information is available and use that information plus any saved state to restore the application's user interface.

For information and guidance on how to implement the methods in your application, see [“Understanding an Application's States and Transitions”](#) (page 29).

Being a Responsible, Multitasking-Aware Application

Applications that run in the background are more limited in what they can do than a foreground application. And even if your application does not run in the background, there are certain guidelines you should follow when implementing your application.

- **Do not make any OpenGL ES calls from your code.** You must not create an `EAGLContext` object or issue any OpenGL ES drawing commands of any kind. Using these calls will cause your application to be terminated immediately.
- **Cancel any Bonjour-related services before being suspended.** When your application moves to the background, and before it is suspended, it should unregister from Bonjour and close listening sockets associated with any network services. A suspended application cannot respond to incoming service requests anyway. Closing out those services prevents them from appearing to be available when they actually are not. If you do not close out Bonjour services yourself, the system closes out those services automatically when your application is suspended.
- **Be prepared to handle connection failures in your network-based sockets.** The system may tear down socket connections while your application is suspended for any number of reasons. As long as your socket-based code is prepared for other types of network failures, such as a lost signal or network transition, this should not lead to any unusual problems. When your application resumes, if it encounters a failure upon using a socket, simply reestablish the connection.
- **Save your application state before moving to the background.** During low-memory conditions, background applications are purged from memory to free up space. Suspended applications are purged first, and no notice is given to the application before it is purged. As a result, before moving to the background, an application should always save enough state information to reconstitute itself later if necessary. Restoring your application to its previous state also provides consistency for the user, who will see a snapshot of your application's main window briefly when it is relaunched.
- **Release any unneeded memory when moving to the background.** Background applications must maintain enough state information to be able to resume operation quickly when they move back to the foreground. But if there are objects or relatively large memory blocks that you no longer need (such as unused images), you should consider releasing that memory when moving to the background.
- **Stop using shared system resources before being suspended.** Applications that interact with shared system resources such as Address Book need to stop using those resources before being suspended. Priority over such resources always goes to the foreground application. At suspend time, if an application is found to be using a shared resource, it will be terminated instead.
- **Avoid updating your windows and views.** While in the background, your application's windows and views are not visible, so you should not try to update them. Although creating and manipulating window and view objects in the background will not cause your application to be terminated, this work should be postponed until your application moves to the foreground.
- **Respond to connect and disconnect notifications for external accessories.** For applications that communicate with external accessories, the system automatically sends a disconnection notification when the application moves to the background. The application must register for this notification and

use it to close out the current accessory session. When the application moves back to the foreground, a matching connection notification is sent giving the application a chance to reconnect. For more information on handling accessory connection and disconnection notifications, see *External Accessory Programming Topics*.

- **Clean up resources for active alerts when moving to the background.** In order to preserve context when switching between applications, the system does not automatically dismiss action sheets (`UIActionSheet`) or alert views (`UIAlertView`) when your application moves to the background. (For applications linked against iOS 3.x and earlier, action sheets and alerts are still dismissed at quit time so that your application's cancellation handler has a chance to run.) It is up to you to provide the appropriate cleanup behavior prior to moving to the background. For example, you might want to cancel the action sheet or alert view programmatically or save enough contextual information to restore the view later (in cases where your application is terminated).
- **Remove sensitive information from views before moving to the background.** When an application transitions to the background, the system takes a snapshot of the application's main window, which it then presents briefly when transitioning your application back to the foreground. Before returning from your `applicationDidEnterBackground:` method, you should hide or obscure passwords and other sensitive personal information that might be captured as part of the snapshot.
- **Do minimal work while running in the background.** The execution time given to background applications is more constrained than the amount of time given to the foreground application. If your application plays background audio or monitors location changes, you should focus on that task only and defer any nonessential tasks until later. Applications that spend too much time executing in the background can be throttled back further by the system or terminated altogether.

If you are implementing a background audio application, or any other type of application that is not suspended while in the background, your application responds to incoming messages in the usual way. In other words, the system may notify your application of low-memory warnings when they occur. And in situations where the system needs to terminate applications to free up even more memory, the application calls its delegate's `applicationWillTerminate:` method to perform any final tasks before exiting.

Initiating Background Tasks

The steps for initiating a background task depend entirely on the task at hand. Some tasks must be initiated explicitly, while others happen in a more automatic way for your application. The following sections provide guidance and examples on how to initiate each type of background task.

Completing a Long-Running Task in the Background

Any time before it is suspended, an application can call the `beginBackgroundTaskWithExpirationHandler:` method to ask the system for extra time to complete some long-running task in the background. If the request is granted, and if the application goes into the background while the task is in progress, the system lets the application run for an additional amount of time instead of suspending it. (The amount of time left for the application to run is available in the `backgroundTimeRemaining` property of the `UIApplication` object.)

You can use background tasks to ensure that important but potentially long-running operations do not end abruptly when the user quits the application. For example, you might use this technique to finish downloading an important file from a network server. There are a couple of design patterns you can use to initiate such tasks:

- Wrap any long-running critical tasks with `beginBackgroundTaskWithExpirationHandler:` and `endBackgroundTask:` calls. This protects those tasks in situations where your application is suddenly moved to the background.
- Wait for your application delegate's `applicationDidEnterBackground:` method to be called and start one or more tasks then.

All calls to the `beginBackgroundTaskWithExpirationHandler:` method must be balanced with a corresponding call to the `endBackgroundTask:` method. The `endBackgroundTask:` method lets the system know that the desired task is complete and that the application can now be suspended. Because applications running in the background have a limited amount of execution time, calling this method is also critical to avoid the termination of your application. (You can always check the value in the `backgroundTimeRemaining` property to see how much time is left.) An application with outstanding background tasks is terminated rather than suspended when its time limit expires. You can also provide an expiration handler for each task and call the `endBackgroundTask:` from there.

An application may have any number of background tasks running at the same time. Each time you start a task, the `beginBackgroundTaskWithExpirationHandler:` method returns a unique identifier for the task. You must pass this same identifier to the `endBackgroundTask:` method when it comes time to end the task.

Listing 4-2 shows how to start a long-running task when your application transitions to the background. In this example, the request to start a background task includes an expiration handler just in case the task takes too long. The task itself is then submitted to a dispatch queue for asynchronous execution so that the `applicationDidEnterBackground:` method can return normally. The use of blocks simplifies the code needed to maintain references to any important variables, such as the background task identifier. The `bgTask` variable is a member variable of the class that stores a pointer to the current background task identifier.

Listing 4-2 Starting a background task at quit time

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    UIApplication* app = [UIApplication sharedApplication];

    bgTask = [app beginBackgroundTaskWithExpirationHandler:^(
        [app endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    )];

    // Start the long-running task and return immediately.
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0), ^{

        // Do the work associated with the task.

        [app endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    });
}
```

In your own expiration handlers, you can include additional code needed to close out your task. However, any code you include should not take long to execute. By the time your expiration handler is called, your application is already very close to its time limit. Doing anything other than clean up your state information and exiting could cause your application to be terminated.

Scheduling the Delivery of Local Notifications

The `UILocalNotification` class in `UIKit` provides a way to schedule the delivery of push notifications locally. Unlike push notifications, which require setting up a remote server, local notifications are scheduled by your application and executed on the current device. You can use this capability to achieve the following behaviors:

- A time-based application can ask the system to post an alert at a specific time in the future. For example, an alarm clock application would use this to implement alarms.
- An application running in the background can post a local notification to get the user's attention.

To schedule the delivery of a local notification, create an instance of the `UILocalNotification` class, configure it, and schedule it using the methods of the `UIApplication` class. The local notification object contains information about the type of notification to deliver (sound, alert, or badge) and the time (when applicable) at which to deliver it. The methods of the `UIApplication` class provide options for delivering notifications immediately or at the scheduled time.

Listing 4-3 shows an example that schedules a single alarm using a date and time that is set by the user. This example supports only one alarm at a time and therefore cancels the previous alarm before scheduling a new one. Your own applications can schedule up to 128 simultaneous notifications, any of which can be configured to repeat at a specified interval. The alarm itself consists of a sound file and an alert box that is played if the application is not running or is in the background when the alarm fires. If the application is active and therefore running in the foreground, the application delegate's `application:didReceiveLocalNotification:` method is called instead.

Listing 4-3 Scheduling an alarm notification

```
- (void)scheduleAlarmForDate:(NSDate*)theDate
{
    UIApplication* app = [UIApplication sharedApplication];
    NSArray* oldNotifications = [app scheduledLocalNotifications];

    // Clear out the old notification before scheduling a new one.
    if ([oldNotifications count] > 0)
        [app cancelAllLocalNotifications];

    // Create a new notification
    UILocalNotification* alarm = [[[UILocalNotification alloc] init] autorelease];
    if (alarm)
    {
        alarm.fireDate = theDate;
        alarm.timeZone = [NSTimeZone defaultTimeZone];
        alarm.repeatInterval = 0;
        alarm.soundName = @"alarmsound.caf";
        alarm.alertBody = @"Time to wake up!";

        [app scheduleLocalNotification:alarm];
    }
}
```

```
}
```

Sound files used with local notifications have the same requirements as those used for push notifications. Custom sound files must be located inside your application's main bundle and support one of the following formats: linear PCM, MA4, uLaw, or aLaw. You can also specify a sound name of `default` to play the default alert sound for the device. When the notification is sent and the sound is played, the system also triggers a vibration on devices that support it.

You can cancel scheduled notifications or get a list of notifications using the methods of the `UIApplication` class. For more information about these methods, see *UIApplication Class Reference*. For additional information about configuring local notifications, see *Local and Push Notification Programming Guide*.

Receiving Location Events in the Background

Applications that use location data to track the user's position now have the option to do so while running in the background. In fact, applications now have more options for receiving location events:

- **Applications can register for significant location changes only. (Recommended)** The significant-change location service is available in iOS 4 and later for devices with a cellular radio. It offers a low-power way to receive location data and is highly recommended. New location updates are provided only when the user's position changes significantly. If the application is suspended while running this service, new location updates will cause the application to be woken up in the background to handle them. Similarly, if the application is terminated while running this service, the system relaunches the application automatically when new location data becomes available.
- **Applications can continue to use the standard location services.** These services are available in all versions of iOS and provide continuous updates while the application is running in the foreground or background. However, if the application is suspended or terminated, new location events do not cause the application to be woken up or relaunched.
- **An application can declare itself as a continuous background location application.** An application that needs the regular location updates offered by the standard location services may declare itself as a continuous background application. It does this by including the `UIBackgroundModes` key in its `Info.plist` file and setting the value of this key to an array containing the `location` string. If an application with this key is running location services when it enters the background, it is not suspended by the system. Instead, it is allowed to continue running so that it may perform location-related tasks in the background.

Regardless of which option you pick, running location services continuously in the background requires enabling the appropriate radio hardware and keeping it active, which can require a significant amount of battery power. If your application does not require precise location information, the best solution is to use the significant location service. You start this service by calling the `startMonitoringSignificantLocationChanges` method of `CLLocationManager`. This service gives the system more freedom to power down the radio hardware and send notifications only when significant location changes are detected. For example, the location manager typically waits until the cell radio detects and starts using a new cell tower for communication.

Another advantage of the significant location service is its ability to wake up and relaunch your application. If a location event arrives while the application is suspended, the application is woken up and given a small amount of time to handle the event. If the application is terminated and a new event arrives, the application is relaunched. Upon relaunch, the options dictionary passed to the

`application:didFinishLaunchingWithOptions:` method contains the `UIApplicationLaunchOptionsLocationKey` key to let the application know that a new location event is available. The application can then restart location services and receive the new event.

For applications that need more precise location data at regular intervals, such as navigation applications, you may need to declare the application as a continuous background application. This option is the least desirable option because it increases power usage considerably. Not only does the application consume power by running in the background, it must also keep location services active, which also consumes power. As a result, you should avoid this option whenever possible.

Regardless of whether you use the significant-change location service or the standard location services, the process for actually receiving location updates is unchanged. After starting location services, the `CLLocationManager:didUpdateToLocation:fromLocation:` method of your location manager delegate is called whenever an event arrives. Of course, applications that run continuously in the background may also want to adjust the implementation of this method to do less work while in the background. For example, you might want to update the application's views only while the application is in the foreground.

For more information about how to use location services in your application, see *Location Awareness Programming Guide*.

Playing Background Audio

Applications that play audio can continue playing that audio while in the background. To indicate that your application plays background audio, include the `UIBackgroundModes` key to its `Info.plist` file. The value for this key is an array containing the `audio` string. When this key is present, the system's audio frameworks automatically prevent your application from being suspended when it moves to the background. Your application continues to run in the background as long as it is playing audio. However, if this key is not present when the application moves to the background, or if your application stops playing audio while in the background, your application is suspended.

You can use any of the system audio frameworks to initiate the playback of background audio and the process for using those frameworks is unchanged. Because your application is not suspended while playing audio, the audio callbacks operate normally while your application is in the background. While running in the background, your application should limit itself to doing only the work necessary to provide audio data for playback. Thus, a streaming audio application would download any new data from its server and push the current audio samples out for playback.

Implementing a VoIP Application

A **Voice over Internet Protocol (VoIP)** application allows the user to make phone calls using an Internet connection instead of the device's cellular service. Such an application needs to maintain a persistent network connection to its associated service. Among other things, this persistent connection allows the application to receive and respond to incoming calls. Rather than keep a VoIP application awake so that it can maintain its network connection, the system provides facilities for managing the sockets associated with that connection while the application is suspended.

To configure a VoIP application, you must do the following:

1. Add the `UIBackgroundModes` key to your application's `Info.plist` file. Set the value of this key to an array that includes the `voip` string.

2. Configure your socket for VoIP usage.
3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to specify the frequency at which your application must be woken to maintain your service.
4. Configure your audio session to handle transitions to and from active use.

Including the `voip` value in the `UIBackgroundModes` key lets the system know that it should allow the application to run in the background as needed to manage its network sockets. An application with this key is also relaunched in the background immediately after system boot to ensure that the VoIP services are always available.

Most VoIP applications also need to be configured as background audio applications in order to process audio while in the background. To do so, you must add both the `audio` and `voip` values to the `UIBackgroundModes` key. Otherwise, there are no changes to how you use the audio frameworks to play and record audio. For more information about the `UIBackgroundModes` key, see *Information Property List Key Reference*.

Configuring Sockets for VoIP Usage

In order for your application to maintain a persistent connection while it is in the background, you must configure the sockets used to communicate with your VoIP service. In iOS, most sockets are managed using higher-level constructs such as streams. As a result, configuration of a socket to support VoIP occurs through the higher-level interfaces. The only thing you have to do beyond the normal configuration is add a special key that tags the interface as being used for a VoIP service. Table 4-1 lists the interfaces that you can configure for VoIP usage and the keys you assign.

Table 4-1 Configuring stream interfaces for VoIP usage

Interface	Configuration
<code>NSInputStream</code> and <code>NSOutputStream</code>	For Cocoa streams, use the <code>setProperty:forKey:</code> method to add the <code>NSStreamNetworkServiceType</code> property to the stream. The value of this property should be set to <code>NSStreamNetworkServiceTypeVoIP</code> .
<code>NSURLRequest</code>	When using the URL loading system, use the <code>setNetworkServiceType:</code> method of your <code>NSMutableURLRequest</code> object to set the network service type of the request. The service type should be set to <code>NSURLNetworkServiceTypeVoIP</code> .
<code>CFReadStreamRef</code> and <code>CFWriteStreamRef</code>	For Core Foundation streams, use the <code>CFReadStreamSetProperty</code> or <code>CFWriteStreamSetProperty</code> function to add the <code>kCFStreamNetworkServiceType</code> property to the stream. The value for this property should be set to <code>kCFStreamNetworkServiceTypeVoIP</code> .

Note: When configuring your sockets, you need to configure only your main signaling channel with the appropriate service type key. You do not need to include this key when configuring your voice channels.

When you configure a stream for VoIP usage, the system takes over management of the underlying socket while your application is suspended. This handoff to the system is transparent to your application. If new data arrives while your application is suspended, the system wakes up your application so that it can process

the data. In the case of an incoming phone call, your application would typically alert the user immediately using a local notification. For other noncritical data, or if the user ignores the call, the system returns your application to the suspended state when it finishes processing the data.

Because VoIP applications need to stay running in order to receive incoming calls, the system automatically relaunches the application if it exits with a nonzero exit code. (This could happen in cases where there is memory pressure and your application is terminated as a result.) However, the system does not maintain your socket connections between different launches of your application. Therefore, at launch time, your application always needs to recreate those connections from scratch.

For more information about configuring Cocoa stream objects, see *Stream Programming Guide*. For information about using URL requests, see *URL Loading System Programming Guide*. And for information about configuring streams using the CFNetwork interfaces, see *CFNetwork Programming Guide*.

Installing a Keep-Alive Handler

To prevent the loss of its connection, a VoIP application typically needs to wake up periodically and check in with its server. To facilitate this behavior, iOS lets you install a special handler using the `setKeepAliveTimeout:handler:` method of `UIApplication`. You typically install this handler when moving to the background. Once installed, the system calls your handler at least once before the timeout interval expires, waking up your application as needed to do so. You can use this handler to check in with your VoIP service and perform any other relevant housekeeping chores.

Your keep-alive handler executes in the background and must return as quickly as possible. Handlers are given a maximum of 30 seconds to perform any needed tasks and return. If a handler has not returned after 30 seconds, the system terminates the application.

When installing your handler, you should specify the largest timeout value that is practical for your needs. The minimum allowable interval for running your handler is 600 seconds and attempting to install a handler with a smaller timeout value will fail. Although the system promises to call your handler block before the timeout value expires, it does not guarantee the exact call time. To improve battery life, the system typically groups the execution of your handler with other periodic system tasks, thereby processing all tasks in one quick burst. As a result, your handler code must be prepared to run earlier than the actual timeout period you specified.

Configuring Your Application's Audio Session

The audio session for a VoIP application must be configured properly to ensure the application works smoothly with other audio-based applications. Because audio playback and recording for a VoIP application are not used all the time, it is especially important that you create and configure your application's audio session only when it is needed to handle a call or provide notifications to the user and release it at other times. This gives other audio applications the opportunity to play their audio the rest of the time.

For information about how to configure and manage an audio session for a VoIP application, see *Audio Session Programming Guide*.

Supporting High-Resolution Screens

Applications built against iOS SDK 4 and later need to be prepared to run on devices with different screen resolutions. Fortunately, iOS makes supporting multiple screen resolutions easy. Most of the work of handling the different types of screens is done for you by the system frameworks. However, your application still needs to do some work to update raster-based images, and depending on your application you may want to do additional work to take advantage of the extra pixels available to you.

Checklist for Supporting High-Resolution Screens

To update your applications for devices with high-resolution screens, you need to do the following:

- Provide a high-resolution image for each image resource in your application bundle, as described in [“Loading Images into Your Application”](#) (page 75).
- Provide high-resolution application and document icons, as described in [“Updating Your Application’s Icons and Launch Images”](#) (page 76).
- For vector-based shapes and content, continue using your custom Core Graphics and UIKit drawing code as before. If you want to add extra detail to your drawn content, see [“Updating Your Custom Drawing Code”](#) (page 76) for information on how to do so.
- If you use Core Animation layers directly, you may need to adjust the scale factor of your layers prior to drawing, as described in [“Accounting for Scale Factors in Core Animation Layers”](#) (page 77).
- If you use OpenGL ES for drawing, decide whether you want to opt in to high-resolution drawing and set the scale factor of your layer accordingly, as described in [“Drawing High-Resolution Content Using OpenGL ES”](#) (page 78).
- For custom images that you create, modify your image-creation code to take the current scale factor into account, as described in [“Creating High-Resolution Bitmap Images Programmatically”](#) (page 76).
- If your application provides the content for Core Animation layers directly, adjust your code as needed to compensate for scale factors, as described in [“Accounting for Scale Factors in Core Animation Layers”](#) (page 77).

Points Versus Pixels

In iOS there is a distinction between the coordinates you specify in your drawing code and the pixels of the underlying device. When using native drawing technologies such as Quartz, UIKit, and Core Animation, you specify coordinate values using a **logical coordinate space**, which measures distances in **points**. This logical coordinate system is decoupled from the **device coordinate space** used by the system frameworks to manage the pixels on the screen. The system automatically maps points in the logical coordinate space to pixels in the device coordinate space, but this mapping is not always one-to-one. This behavior leads to an important fact that you should always remember:

One point does not necessarily correspond to one pixel on the screen.

The purpose of using points (and the logical coordinate system) is to provide a fixed frame of reference for drawing. The actual size of a point is irrelevant. The goal of points is to provide a relatively consistent scale that you can use in your code to specify the size and position of views and rendered content. How points are actually mapped to pixels is a detail that is handled by the system frameworks. For example, on a device with a high-resolution screen, a line that is one point wide may actually result in a line that is two pixels wide on the screen. The result is that if you draw the same content on two similar devices, with only one of them having a high-resolution screen, the content appears to be about the same size on both devices.

In your own drawing code, you use points most of the time, but there are times when you might need to know how points are mapped to pixels. For example, on a high-resolution screen, you might want to use the extra pixels to provide extra detail in your content, or you might simply want to adjust the position or size of content in subtle ways. In iOS 4 and later, the `UIScreen`, `UIView`, `UIImage`, and `CALayer` classes expose a **scale factor** that tells you the relationship between points and pixels for that particular object. Before iOS 4, this scale factor was assumed to be 1.0, but in iOS 4 and later it may be either 1.0 or 2.0, depending on the resolution of the underlying device. In the future, other scale factors may also be possible.

Drawing Improvements That You Get for Free

The drawing technologies in iOS provide a lot of support to help you make your rendered content look good regardless of the resolution of the underlying screen:

- Standard UIKit views (text views, buttons, table views, and so on) automatically render correctly at any resolution.
- Vector-based content (`UIBezierPath`, `CGPathRef`, PDF) automatically takes advantage of any additional pixels to render sharper lines for shapes.
- Text is automatically rendered sharper at higher resolutions.
- UIKit supports the automatic loading of high-resolution variants (@2x) of your images.

The reason most of your existing drawing code just works is that native drawing technologies such as Core Graphics take the current scale factor into account for you. For example, if one of your views implements a `drawRect:` method, UIKit automatically sets the scale factor for that view to the screen's scale factor. In addition, UIKit automatically modifies the **current transformation matrix (CTM)** of any graphics contexts used during drawing to take into account the view's scale factor. Thus, any content you draw in your `drawRect:` method is scaled appropriately for the underlying device's screen.

If your application uses only native drawing technologies for its rendering, the only thing you need to do is provide high-resolution versions of your images. Applications that use nothing but system views or that rely solely on vector-based content do not need to be modified. But applications that use images need to provide new versions of those images at the higher resolution. Specifically, you must scale your images by a factor of two, resulting in twice as many pixels horizontally and vertically as before and four times as many pixels overall. For more information on updating your image resources, see [“Updating Your Image Resource Files”](#) (page 75).

Updating Your Image Resource Files

Applications running in iOS 4 should now include two separate files for each image resource. One file provides a standard-resolution version of a given image, and the second provides a high-resolution version of the same image. The naming conventions for each pair of image files is as follows:

- Standard: `<ImageName><device_modifier>.<filename_extension>`
- High resolution: `<ImageName>@2x<device_modifier>.<filename_extension>`

The `<ImageName>` and `<filename_extension>` portions of each name specify the usual name and extension for the file. The `<device_modifier>` portion is optional and contains either the string `~ipad` or `~iphone`. You include one of these modifiers when you want to specify different versions of an image for iPad and iPhone. The inclusion of the `@2x` modifier for the high-resolution image is new and lets the system know that the image is the high-resolution variant of the standard image.

Important: When creating high-resolution versions of your images, place the new versions in the same location in your application bundle as the original.

Loading Images into Your Application

The `UIImage` class handles all of the work needed to load high-resolution images into your application. When creating new image objects, you use the same name to request both the standard and the high-resolution versions of your image. For example, if you have two image files, named `Button.png` and `Button@2x.png`, you would use the following code to request your button image:

```
UIImage* anImage = [UIImage imageNamed:@"Button"];
```

Note: In iOS 4 and later, you may omit the filename extension when specifying image names.

On devices with high-resolution screens, the `imageNamed:`, `imageWithContentsOfFile:`, and `initWithContentsOfFile:` methods automatically look for a version of the requested image with the `@2x` modifier in its name. If it finds one, it loads that image instead. If you do not provide a high-resolution version of a given image, the image object still loads a standard-resolution image (if one exists) and scales it during drawing.

When it loads an image, a `UIImage` object automatically sets the `size` and `scale` properties to appropriate values based on the suffix of the image file. For standard resolution images, it sets the `scale` property to 1.0 and sets the size of the image to the image's pixel dimensions. For images with the `@2x` suffix in the filename, it sets the `scale` property to 2.0 and halves the width and height values to compensate for the scale factor. These halved values correlate correctly to the point-based dimensions you need to use in the logical coordinate space to render the image.

Note: If you use Core Graphics to create an image, remember that Quartz images do not have an explicit scale factor, so their scale factor is assumed to be 1.0. If you want to create a `UIImage` object from a `CGImageRef` data type, use the `initWithCGImage:scale:orientation:` to do so. That method allows you to associate a specific scale factor with your Quartz image data.

A `UIImage` object automatically takes its scale factor into account during drawing. Thus, any code you have for rendering images should work the same as long as you provide the correct image resources in your application bundle.

Using an Image View to Display Images

If your application uses a `UIImageView` object to present images, all of the images you assign to that view must use the same scale factor. You can use an image view to display a single image or to animate several images, and you can also provide a highlight image. Therefore, if you provide high-resolution versions for one of these images, then all must have high-resolution versions as well.

Updating Your Application's Icons and Launch Images

In addition to updating your application's custom image resources, you should also provide new high-resolution icons for your application's icon and launch images. The process for updating these image resources is the same as for all other image resources. Create a new version of the image, add the `@2x` modifier string to the corresponding image filename, and treat the image as you do the original. For example, for application icons, add the high-resolution image filename to the `CFBundleIconFiles` key of your application's `Info.plist` file.

For information about specifying the icons for your application, see [“Application Icons”](#) (page 100). For information about specifying launch images, see [“Application Launch Images”](#) (page 101).

Updating Your Custom Drawing Code

When you do any custom drawing in your application, most of the time you should not need to care about the resolution of the underlying screen. The native drawing technologies automatically ensure that the coordinates you specify in the logical coordinate space map correctly to pixels on the underlying screen. Sometimes, however, you might need to know what the current scale factor is in order to render your content correctly. For those situations, UIKit, Core Animation, and other system frameworks provide the help you need to do your drawing correctly.

Creating High-Resolution Bitmap Images Programmatically

If you currently use the `UIGraphicsBeginImageContext` function to create bitmaps, you may want to adjust your code to take scale factors into account. The `UIGraphicsBeginImageContext` function always creates images with a scale factor of 1.0. If the underlying device has a high-resolution screen, an image created with this function might not appear as smooth when rendered. To create an image with a scale factor other than 1.0, use the `UIGraphicsBeginImageContextWithOptions` instead. The process for using this function is the same as for the `UIGraphicsBeginImageContext` function:

1. Call `UIGraphicsBeginImageContextWithOptions` to create a bitmap context (with the appropriate scale factor) and push it on the graphics stack.
2. Use UIKit or Core Graphics routines to draw the content of the image.
3. Call `UIGraphicsGetImageFromCurrentImageContext` to get the bitmap's contents.
4. Call `UIGraphicsEndImageContext` to pop the context from the stack.

For example, the following code snippet creates a bitmap that is 200 x 200 pixels. (The number of pixels is determined by multiplying the size of the image by the scale factor.)

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(100.0,100.0), NO, 2.0);
```

Note: If you always want the bitmap to be scaled appropriately for the main screen of the device, set the scale factor to 0.0 when calling the `UIGraphicsBeginImageContextWithOptions` function.

Tweaking Native Content for High-Resolution Screens

If you want to draw your content differently on high-resolution screens, you can use the current scale factor to modify your drawing code. For example, suppose you have a view that draws a 1-pixel-wide border around its edge. On devices where the scale factor is 2.0, using a `UIBezierPath` object to draw a line with a width of 1.0 would result in a line that was 2 pixels wide. In this case, you could divide your line width by the scale factor to obtain a proper 1-pixel-wide line.

Of course, changing drawing characteristics based on scale factor may have unexpected consequences. A 1-pixel-wide line might look nice on some devices but on a high-resolution device might be so thin that it is difficult to see clearly. It is up to you to determine whether to make such a change.

Accounting for Scale Factors in Core Animation Layers

Applications that use Core Animation layers directly to provide content may need to adjust their drawing code to account for scale factors. Normally, when you draw in your view's `drawRect:` method, or in the `drawLayer:inContext:` method of the layer's delegate, the system automatically adjusts the graphics context to account for scale factors. However, knowing or changing that scale factor might still be necessary when your view does one of the following:

- Creates additional Core Animation layers with different scale factors and composites them into its own content
- Sets the `contents` property of a Core Animation layer directly

Core Animation's compositing engine looks at the `contentsScale` property of each layer to determine whether the contents of that layer need to be scaled during compositing. If your application creates layers without an associated view, each new layer object's scale factor is set to 1.0 initially. If you do not change that scale factor, and if you subsequently draw the layer on a high-resolution screen, the layer's contents are scaled automatically to compensate for the difference in scale factors. If you do not want the contents to be scaled, you can change the layer's scale factor to 2.0, but if you do so without providing high-resolution content, your existing content may appear smaller than you were expecting. To fix that problem, you need to provide higher-resolution content for your layer.

Important: The `contentsGravity` property of the layer plays a role in determining whether standard-resolution layer content is scaled on a high-resolution screen. This property is set to the value `kCAGravityResize` by default, which causes the layer content to be scaled to fit the layer's bounds. Changing the gravity to a nonresizing option eliminates the automatic scaling that would otherwise occur. In such a situation, you may need to adjust your content or the scale factor accordingly.

Adjusting the content of your layer to accommodate different scale factors is most appropriate when you set the `contents` property of a layer directly. Quartz images have no notion of scale factors and therefore work directly with pixels. Therefore, before creating the `CGImageRef` object you plan to use for the layer's contents, check the scale factor and adjust the size of your image accordingly. Specifically, load an appropriately sized image from your application bundle or use the `UIGraphicsBeginImageContextWithOptions` function to create an image whose scale factor matches the scale factor of your layer. If you do not create a high-resolution bitmap, the existing bitmap may be scaled as discussed previously.

For information on how to specify and load high-resolution images, see [“Updating Your Image Resource Files”](#) (page 75). For information about how to create high-resolution images, see [“Creating High-Resolution Bitmap Images Programmatically”](#) (page 76).

Drawing High-Resolution Content Using OpenGL ES

If your application uses OpenGL ES for rendering, your existing drawing code should continue to work without any changes. When drawn on a high-resolution screen, though, your content is scaled accordingly and will appear more blocky. The reason for the blocky appearance is that the default behavior of the `CAEAGLLayer` class, which you use to back your OpenGL ES renderbuffers, is the same as other Core Animation layer objects. In other words, its scale factor is set to 1.0 initially, which causes the Core Animation compositor to scale the contents of the layer on high-resolution screens. To avoid this blocky appearance, you need to increase the size of your OpenGL ES renderbuffers to match the size of the screen. (With more pixels, you can then increase the amount of detail you provide for your content.) Because adding more pixels to your renderbuffers has performance implications, though, you must explicitly opt in to support high-resolution screens.

To enable high-resolution drawing, you must change the scale factor of the view you use to present your OpenGL ES content. Changing the `contentScaleFactor` property of your view from 1.0 to 2.0 triggers a matching change to the scale factor of the underlying `CAEAGLLayer` object. The `renderbufferStorage:fromDrawable:` method, which you use to bind the layer object to your renderbuffers, calculates the size of the renderbuffer by multiplying the layer's bounds by its scale factor. Thus, doubling the scale factor doubles the width and height of the resulting renderbuffer, giving you more pixels for your content. After that, it is up to you to provide the content for those additional pixels.

Listing 5-1 shows the proper way to bind your layer object to your renderbuffers and retrieve the resulting size information. If you used the OpenGL ES application template to create your code, then this step is already done for you, and the only thing you need to do is set the scale factor of your view appropriately. If you did not use the OpenGL ES application template, you should use code similar to this to retrieve the renderbuffer size. You should never assume that the renderbuffer size is fixed for a given type of device.

Listing 5-1 Initializing a renderbuffer's storage and retrieving its actual dimensions

```
GLuint colorRenderbuffer;
glGenRenderbuffersOES(1, &colorRenderbuffer);
glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
[myContext renderbufferStorage:GL_RENDERBUFFER_OES fromDrawable:myEAGLLayer];
```

```
// Get the renderbuffer size.  
GLint width;  
GLint height;  
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_WIDTH_OES,  
    &width);  
glGetRenderbufferParameterivOES(GL_RENDERBUFFER_OES, GL_RENDERBUFFER_HEIGHT_OES,  
    &height);
```

Important: A view that is backed by a `CAEAGLLayer` object should not implement a custom `drawRect:` method. Implementing a `drawRect:` method causes the system to change the default scale factor of the view so that it matches the scale factor of the screen. If your drawing code is not expecting this behavior, your application content will not be rendered correctly.

If you do opt in to high-resolution drawing, you also need to adjust the model and texture assets of your application accordingly. For example, when running on an iPad or a high-resolution device, you might want to choose larger models and more detailed textures to take advantage of the increased number of pixels. Conversely, on a standard-resolution iPhone, you can continue to use smaller models and textures.

An important factor when determining whether to support high-resolution content is performance. The quadrupling of pixels that occurs when you change the scale factor of your layer from 1.0 to 2.0 puts additional pressure on the fragment processor. If your application performs many per-fragment calculations, the increase in pixels may reduce your application's frame rate. If you find your application runs significantly slower at the higher scale factor, consider one of the following options:

- Optimize your fragment shader's performance using the performance-tuning guidelines found in *OpenGL ES Programming Guide for iOS*.
- Choose a simpler algorithm to implement in your fragment shader. By doing so, you are reducing the quality of each individual pixel to render the overall image at a higher resolution.
- Use a fractional scale factor between 1.0 and 2.0. A scale factor of 1.5 provides better quality than a scale factor of 1.0 but needs to fill fewer pixels than an image scaled to 2.0.
- OpenGL ES in iOS 4 and later offers multisampling as an option. Even though your application can use a smaller scale factor (even 1.0), implement multisampling anyway. An added advantage is that this technique also provides higher quality on devices that do not support high-resolution displays.

The best solution depends on the needs of your OpenGL ES application; you should test more than one of these options and choose the approach that provides the best balance between performance and image quality.

Implementing Application Preferences

In traditional desktop applications, preferences are application-specific settings used to configure the behavior or appearance of an application. iOS also supports application preferences, although not as an integral part of your application. Instead of each application displaying a custom user interface for its preferences, all application-level preferences are displayed using the system-supplied Settings application.

In order to integrate your custom application preferences into the Settings application, you must include a specially formatted settings bundle in the top-level directory of your application bundle. This settings bundle provides information about your application preferences to the Settings application, which is then responsible for displaying those preferences and updating the preferences database with any user-supplied values. At runtime, your application retrieves these preferences using the standard retrieval APIs. The sections that follow describe both the format of the settings bundle and the APIs you use to retrieve your preferences values.

Guidelines for Preferences

Adding your application preferences to the Settings application is most appropriate for productivity-style applications and in situations where you have preference values that are typically configured once and then rarely changed. For example, the Mail application uses these preferences to store the user's account information and message-checking settings. Because the Settings application has support for displaying preferences hierarchically, manipulating your preferences from the Settings application is also more appropriate when you have a large number of preferences. Providing the same set of preferences in your application might require too many screens and might cause confusion for the user.

When your application has only a few options or has options that the user might want to change regularly, you should think carefully about whether the Settings application is the right place for them. For instance, utility applications provide custom configuration options on the back of their main view. A special control on the view flips it over to display the options, and another control flips the view back. For simple applications, this type of behavior provides immediate access to the application's options and is much more convenient for the user than going to Settings.

For games and other full-screen applications, you can use the Settings application or implement your own custom screens for preferences. Custom screens are often appropriate in games because those screens are treated as part of the game's setup. You can also use the Settings application for your preferences if you think it is more appropriate for your game flow.

Note: You should never spread your preferences across the Settings application and custom application screens. For example, a utility application with preferences on the back side of its main view should not also have configurable preferences in the Settings application. If you have preferences, pick one solution and use it exclusively.

If you do incorporate settings into the Settings application, be sure you respond to the `NSUserDefaultsDidChangeNotification` notification as described in “[Responding to Changes in Your Application’s Settings](#)” (page 38).

The Preferences Interface

The Settings application implements a hierarchical set of pages for navigating application preferences. The main page of the Settings application displays the system and third-party applications whose preferences can be customized. Selecting a third-party application takes the user to the preferences for that application.

Each application has at least one page of preferences, referred to as the *main page*. If your application has only a few preferences, the main page may be the only one you need. If the number of preferences gets too large to fit on the main page, however, you can add more pages. These additional pages become child pages of the main page. The user accesses them by tapping on a special type of preference, which links to the new page.

Each preference you display must be of a specific type. The type of the preference defines how the Settings application displays that preference. Most preference types identify a particular type of control that is used to set the preference value. Some types provide a way to organize preferences, however. Table 6-1 lists the different element types supported by the Settings application and points out how you might use each type to implement your own preference pages.

Table 6-1 Preference element types

Element Type	Description
Text Field	The text field type displays an optional title and an editable text field. You can use this type for preferences that require the user to specify a custom string value. The key for this type is <code>PSTextFieldSpecifier</code> .
Title	The title type displays a read-only string value. You can use this type to display read-only preference values. (If the preference contains cryptic or nonintuitive values, this type lets you map the possible values to custom strings.) The key for this type is <code>PSTitleValueSpecifier</code> .
Toggle Switch	The toggle switch type displays an ON/OFF toggle button. You can use this type to configure a preference that can have only one of two values. Although you typically use this type to represent preferences containing Boolean values, you can also use it with preferences containing non-Boolean values. The key for this type is <code>PSToggleSwitchSpecifier</code> .

Element Type	Description
Slider	The slider type displays a slider control. You can use this type for a preference that represents a range of values. The value for this type is a real number whose minimum and maximum value you specify. The key for this type is <code>PSSliderSpecifier</code> .
Multi Value	The multi value type lets the user select one value from a list of values. You can use this type for a preference that supports a set of mutually exclusive values. The values can be of any type. The key for this type is <code>PSMultiValueSpecifier</code> .
Group	The group type is a way for you to organize groups of preferences on a single page. The group type does not represent a configurable preference. It simply contains a title string that is displayed immediately before one or more configurable preferences. The key for this type is <code>PSGroupSpecifier</code> .
Child Pane	The child pane type lets the user navigate to a new page of preferences. You use this type to implement hierarchical preferences. For more information on how you configure and use this preference type, see “Hierarchical Preferences” (page 85). The key for this type is <code>PSChildPaneSpecifier</code> .

For detailed information about the format of each preference type, see *Settings Application Schema Reference*. To learn how to create and edit Setting page files, see [“Adding and Modifying the Settings Bundle”](#) (page 86).

The Settings Bundle

In iOS, you specify your application’s preferences through a special settings bundle. This bundle has the name `Settings.bundle` and resides in the top-level directory of your application’s bundle. This bundle contains one or more Settings Page files that provide detailed information about your application’s preferences. It may also include other support files needed to display your preferences, such as images or localized strings. Table 6-2 lists the contents of a typical settings bundle.

Table 6-2 Contents of the `Settings.bundle` directory

Item name	Description
<code>Root.plist</code>	The Settings Page file containing the preferences for the root page. The contents of this file are described in more detail in “The Settings Page File Format” (page 84).
Additional <code>.plist</code> files.	If you build a set of hierarchical preferences using child panes, the contents for each child pane are stored in a separate Settings Page file. You are responsible for naming these files and associating them with the correct child pane.

Item name	Description
One or more <code>.lproj</code> directories	These directories store localized string resources for your Settings Page files. Each directory contains a single strings file, whose title is specified in your Settings Page. The strings files provide the localized content to display to the user for each of your preferences.
Additional images	If you use the slider control, you can store the images for your slider in the top-level directory of the bundle.

In addition to the settings bundle, your application bundle can contain a custom icon for your application settings. The Settings application displays the icon you provide next to the entry for your application preferences. For information about application icons and how you specify them, see [“Application Icons”](#) (page 100).

When the Settings application launches, it checks each custom application for the presence of a settings bundle. For each custom bundle it finds, it loads that bundle and displays the corresponding application's name and icon in the Settings main page. When the user taps the row belonging to your application, Settings loads the `Root.plist` Settings Page file for your settings bundle and uses that file to display your application's main page of preferences.

In addition to loading your bundle's `Root.plist` Settings Page file, the Settings application also loads any language-specific resources for that file, as needed. Each Settings Page file can have an associated `.strings` file containing localized values for any user-visible strings. As it prepares your preferences for display, the Settings application looks for string resources in the user's preferred language and substitutes them into your preferences page prior to display.

The Settings Page File Format

Each Settings Page file in your settings bundle is stored in the iPhone Settings property-list file format, which is a structured file format. The simplest way to edit Settings Page files is to use the built-in editor facilities of Xcode; see [“Preparing the Settings Page for Editing”](#) (page 87). You can also edit property-list files using the Property List Editor application that comes with the Xcode tools.

Note: Xcode automatically converts any XML-based property files in your project to binary format when building your application. This conversion saves space and is done for you automatically at build time.

The root element of each Settings Page file contains the keys listed in Table 6-3. Only one key is actually required, but it is recommended that you include both of them.

Table 6-3 Root-level keys of a preferences Settings Page file

Key	Type	Value
<code>PreferenceSpecifiers</code> (required)	Array	The value for this key is an array of dictionaries, with each dictionary containing the information for a single preference element. For a list of element types, see Table 6-1 (page 82). For a description of the keys associated with each element type, see <i>Settings Application Schema Reference</i> .

Key	Type	Value
StringsTable	String	The name of the strings file associated with this file. A copy of this file (with appropriate localized strings) should be located in each of your bundle's language-specific project directories. If you do not include this key, the strings in this file are not localized. For information on how these strings are used, see “Localized Resources” (page 86).

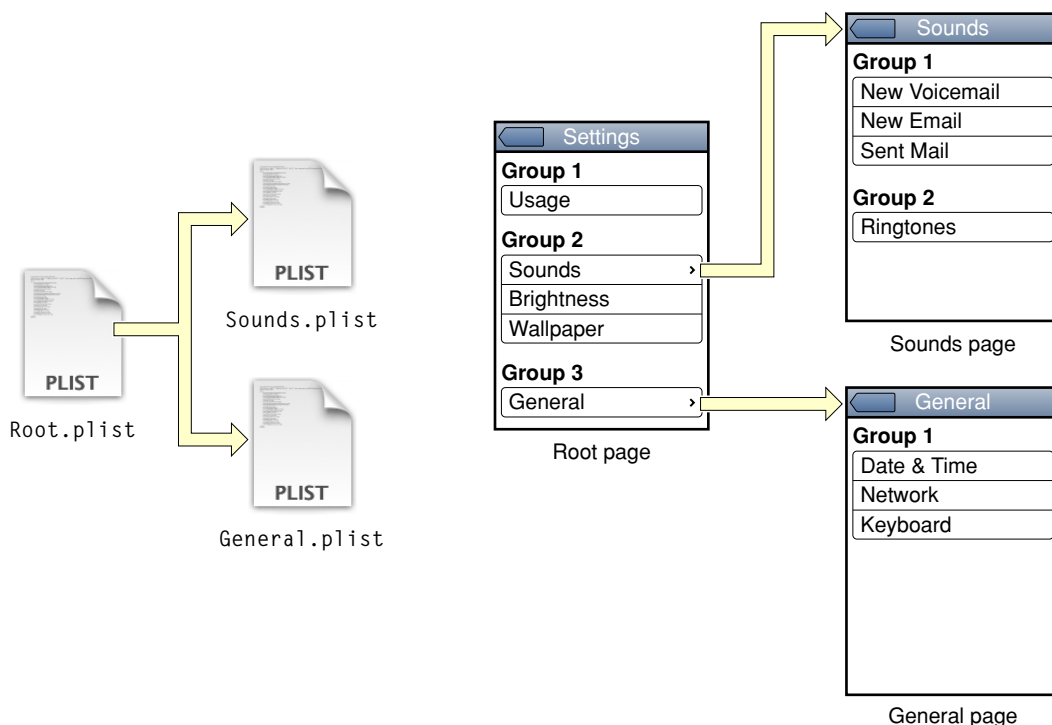
Hierarchical Preferences

If you plan to organize your preferences hierarchically, each page you define must have its own separate `.plist` file. Each `.plist` file contains the set of preferences displayed only on that page. Your application's main preferences page is always stored in the `Root.plist` file. Additional pages can be given any name you like.

To specify a link between a parent page and a child page, you include a child pane element in the parent page. A child pane element creates a row that, when tapped, displays a new page of settings. The `File` key of the child pane element identifies the name of the `.plist` file that defines the contents of the child page. The `Title` key identifies the title of the child page; this title is also used as the text of the row that the user taps to display the child page. The Settings application automatically provides navigation controls on the child page to allow the user to navigate back to the parent page.

Figure 6-1 shows how this hierarchical set of pages works. The left side of the figure shows the `.plist` files, and the right side shows the relationships between the corresponding pages.

Figure 6-1 Organizing preferences using child panes



For more information about child pane elements and their associated keys, see *Settings Application Schema Reference*.

Localized Resources

Because preferences contain user-visible strings, you should provide localized versions of those strings with your settings bundle. Each page of preferences can have an associated `.strings` file for each localization supported by your bundle. When the Settings application encounters a key that supports localization, it checks the appropriately localized `.strings` file for a matching key. If it finds one, it displays the value associated with that key.

When looking for localized resources such as `.strings` files, the Settings application follows the same rules that Mac OS X applications do. It first tries to find a localized version of the resource that matches the user's preferred language setting. If a resource does not exist for the user's preferred language, an appropriate fallback language is selected.

For information about the format of strings files, language-specific project directories, and how language-specific resources are retrieved from bundles, see *Internationalization Programming Topics*.

Adding and Modifying the Settings Bundle

Xcode provides a template for adding a Settings bundle to your current project. The default settings bundle contains a `Root.plist` file and a default language directory for storing any localized resources. You can then expand this bundle to include additional property list files and resources needed by your settings bundle.

Adding the Settings Bundle

To add a settings bundle to your Xcode project:

1. Choose File > New File.
2. Choose the iOS > Resource > Settings Bundle template.
3. Name the file `Settings.bundle`.

In addition to adding a new settings bundle to your project, Xcode automatically adds that bundle to the Copy Bundle Resources build phase of your application target. Thus, all you have to do is modify the property list files of your settings bundle and add any needed resources.

The newly added `Settings.bundle` bundle has the following structure:

```
Settings.bundle/  
  Root.plist  
  en.lproj/  
    Root.strings
```

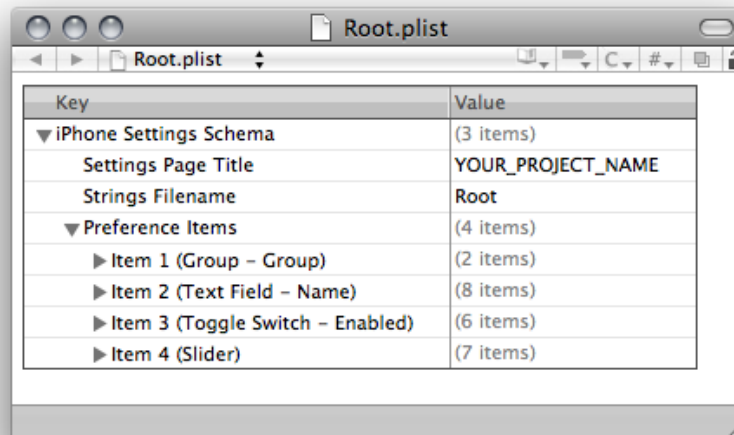
Preparing the Settings Page for Editing

After creating your settings bundle using the Settings Bundle template, you can format the contents of your schema files to make them easier to edit. The following steps show you how to do this for the `Root.plist` file of your settings bundle but the steps are the same for any other schema files you create.

1. Display the contents of the `Root.plist` file of your settings bundle.
 - a. In the Groups & Files list, disclose `Settings.bundle` to view its contents.
 - b. Select the `Root.plist` file. Its contents appear in the Detail view.
2. In the Detail view, select the Root key of the `Root.plist` file.
3. Choose View > Property List Type > iPhone Settings plist.

This command formats the contents of the property list inside the Detail view. Instead of showing the property list key names and values, Xcode substitutes human-readable strings (as shown in Figure 6-2) to make it easier for you to understand and edit the file's contents.

Figure 6-2 Formatted contents of the `Root.plist` file



Key	Value
▼ iPhone Settings Schema	(3 items)
Settings Page Title	YOUR_PROJECT_NAME
Strings Filename	Root
▼ Preference Items	(4 items)
▶ Item 1 (Group – Group)	(2 items)
▶ Item 2 (Text Field – Name)	(8 items)
▶ Item 3 (Toggle Switch – Enabled)	(6 items)
▶ Item 4 (Slider)	(7 items)

Configuring a Settings Page: A Tutorial

This section contains a tutorial that shows you how to configure a Settings page to display the controls you want. The goal of the tutorial is to create a page like the one in Figure 6-3. If you have not yet created a settings bundle for your project, you should do so as described in [“Preparing the Settings Page for Editing”](#) (page 87) before proceeding with these steps.

Figure 6-3 A root Settings page

1. Change the value of the Settings Page Title key to the name of your application.
Double-click the `YOUR_PROJECT_NAME` text and change the text to `MyApp`.
2. Disclose the Preference Items key to display the default items that come with the template.
3. Change the title of Item 1 to Sound.
 - Disclose Item 1 of Preference Items.
 - Change the value of the Title key from Group to Sound.
 - Leave the Type key set to Group.
4. Create the first toggle switch for the newly renamed Sound group.
 - Select Item 3 of Preference Items and choose Edit > Cut.
 - Select Item 1 and choose Edit > Paste. (This moves the toggle switch item in front of the text field item.)
 - Disclose the toggle switch item to reveal its configuration keys.
 - Change the value of the Title key to Play Sounds.
 - Change the value of the Identifier key to `play_sounds_preference`. The item should now be configured as shown in the following figure.

▼ Item 2 (Toggle Switch – Play Sounds) ▼ (6 items)	
Type	Toggle Switch
Title	Play Sounds
Identifier	play_sounds_preference
Default Value	<input checked="" type="checkbox"/>
Value for ON	YES
Value for OFF	NO

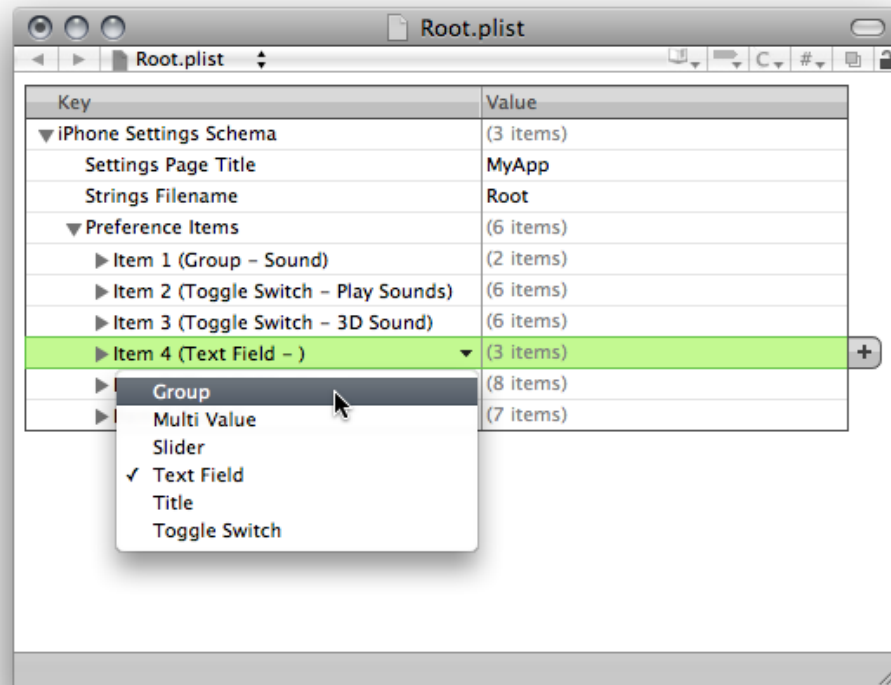
5. Create a second toggle switch for the Sound group.

- Select `Item 2` (the Play Sounds toggle switch).
- Select `Edit > Copy`.
- Select `Edit > Paste`. This places a copy of the toggle switch right after the first one.
- Disclose the new toggle switch item to reveal its configuration keys.
- Change the value of its `Title` key to `3D Sound`.
- Change the value of its `Identifier` key to `3D_sound_preference`.

At this point, you have finished the first group of settings and are ready to create the User Info group.

6. Change `Item 4` into a Group element and name it `User Info`.

- Click `Item 4` in the `Preferences Items`. This displays a drop-down menu with a list of item types.
- From the drop-down menu, select `Group` to change the type of the element.



- Disclose the contents of Item 4.
- Set the value of the Title key to User Info.

7. Create the Name field.

- Select Item 5 in the Preferences Item.
- Using the drop-down menu, change its type to Text Field.
- Set the value of the Title key to User Info.
- Set value of the Identifier key to user_name.
- Toggle the disclosure button of the item to hide its contents.

8. Create the Experience Level settings.

- Select Item 5 and click the plus (+) button (or press Return) to create a new item.
- Click the new item and set its type to Multi Value.
- Disclose the items contents and set its title to Experience Level, its identifier to experience_preference, and its default value to 0.
- With the Default Value key selected, click the plus button to add a Titles array.
- Open the disclosure button for the Titles array and click the items button along the right edge of the table. Clicking this button adds a new subitem to Titles.

- Select the new subitem and click the plus button 2 more times to create 3 total subitems.
 - Set the values of the subitems to `Beginner`, `Expert`, and `Master`.
 - Select the `Titles` key again and click its disclosure button to hide its subitems.
 - Click the plus button to create the `Values` array.
 - Add 3 subitems to the `Values` array and set their values to 0, 1, and 2.
 - Click the disclosure button of `Item 6` to hide its contents.
9. Add the final group to your settings page.
- Create a new item and set its type to `Group` and its title to `Gravity`.
 - Create another new item and set its type to `Slider`, its identifier to `gravity_preference`, its default value to 1, and its maximum value to 2.

Creating Additional Settings Page Files

The Settings Bundle template includes the `Root.plist` file, which defines your application's top Settings page. To define additional Settings pages, you must add additional property list files to your settings bundle. You can do this either from the Finder or from Xcode.

To add a property list file to your settings bundle in Xcode, do the following:

1. In the Groups and Files pane, open your settings bundle and select the `Root.plist` file.
2. Choose `File > New`.
3. Choose `Other > Property List`.
4. Select the new file and choose `View > Property List Type > iPhone Settings plist` to configure it as a settings file.

After adding a new Settings page to your settings bundle, you can edit the page's contents as described in [“Configuring a Settings Page: A Tutorial”](#) (page 87). To display the settings for your page, you must reference it from a `Child Pane` element as described in [“Hierarchical Preferences”](#) (page 85).

Accessing Your Preferences

iOS applications get and set preferences values using either the Foundation or Core Foundation frameworks. In the Foundation framework, you use the `NSUserDefaults` class to get and set preference values. In the Core Foundation framework, you use several preferences-related functions to get and set values.

Listing 6-1 shows a simple example of how to read a preference value from your application. This example uses the `NSUserDefaults` class to read a value from the preferences created in [“Configuring a Settings Page: A Tutorial”](#) (page 87) and assign it to an application-specific instance variable.

Listing 6-1 Accessing preference values in an application

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [self setShouldPlaySounds:[defaults boolForKey:@"play_sounds_preference"]];

    // Finish app initialization...
}
```

For information about the `NSUserDefaults` methods used to read and write preferences, see *NSUserDefaults Class Reference*. For information about the Core Foundation functions used to read and write preferences, see *Preferences Utilities Reference*.

Debugging Preferences for Simulated Applications

When running your application, the iPhone Simulator stores any preferences values for your application in `~/Library/Application Support/iPhone Simulator/User/Applications/<APP_ID>/Library/Preferences`, where `<APP_ID>` is a programmatically generated directory name that iOS uses to identify your application.

Each time you reinstall your application, iOS performs a clean install, which deletes any previous preferences. In other words, building or running your application from Xcode always installs a new version, replacing any old contents. To test preference changes between successive executions, you must run your application directly from the simulator interface and not from Xcode.

Build-Time Configuration Details

The configuration of your application is an important part of the development process. An iOS application uses a structured directory to manage its code and resource files. And although most of the files in this directory are custom files to support your application, some are resource files required by the system to run your application. Configuring these files correctly is therefore crucial to creating your application.

The Application Bundle

When you build your iOS application, Xcode packages it as a bundle. A **bundle** is a directory in the file system that groups related resources together in one place. An iOS application bundle contains the application executable and any resources used by the application (for instance, the application icon, other images, and localized content). Table 7-1 lists the contents of a typical iOS application bundle, which for demonstration purposes here is called `MyApp`. This example is for illustrative purposes only. Some of the files listed in this table may not appear in your own application bundles.

Table 7-1 A typical application bundle

Files	Description
<code>MyApp</code>	The executable file containing your application's code. The name of this file is the same as your application name minus the <code>.app</code> extension. This file is required.
<code>Info.plist</code>	Also known as the information property list file, a file defining key values for the application, such as bundle ID, version number, and display name. This file is required. For more information about this file, see “The Information Property List” (page 95).
<code>MainWindow.nib</code>	The application's main nib file, containing the default interface objects to load at application launch time. Typically, this nib file contains the application's main window object and an instance of the application delegate object. Other interface objects are then either loaded from additional nib files or created programmatically by the application. (The name of the main nib file can be changed by assigning a different value to the <code>NSMainNibFile</code> key in the <code>Info.plist</code> file. See “The Information Property List” (page 95) for further information.)
Application icons	One or more image files containing the icons used to represent your application. Differently sized variants of the application icon may also be provided. An application icon is required. For information about these image files, see “Application Icons” (page 100).

Files	Description
One or more launch images	<p>A screen-sized image to display when your application is launched. The system uses this file as a temporary background until your application loads its window and user interface.</p> <p>At least one launch image is required. For information about specifying launch images, see “Application Launch Images” (page 101).</p>
iTunesArtwork	<p>The 512 x 512 pixel icon for an application that is distributed using ad hoc distribution. This icon would normally be provided by the App Store; because applications distributed in an ad hoc manner do not go through the App Store, however, it must be present in the application bundle instead. iTunes uses this icon to represent your application. (The file you specify should be the same one you would have submitted to the App Store (typically a JPEG or PNG file), were you to distribute your application that way. The filename must be the one shown at left and must not include a filename extension.)</p>
Settings.bundle	<p>A file package that you use to add application preferences to the Settings application. This bundle contains property lists and other resource files to configure and display your preferences. For more information about specifying settings, see “Displaying Application Preferences” (page 60).</p>
sun.png (or other resource files)	<p>Nonlocalized resources are placed at the top level of the bundle directory (sun.png represents a nonlocalized image file in the example). The application uses nonlocalized resources regardless of the language setting chosen by the user.</p>
en.lproj fr.lproj es.lproj other language-specific project directories	<p>Localized resources are placed in subdirectories with an ISO 639-1 language abbreviation for a name plus an .lproj suffix. (For example, the en.lproj, fr.lproj, and es.lproj directories contain resources localized for English, French, and Spanish.) For more information, see “Internationalizing Your Application” (page 104).</p>

An iOS application should be internationalized and have a *language.lproj* directory for each language it supports. In addition to providing localized versions of your application’s custom resources, you can also localize your application icon, launch images, and Settings icon by placing files with the same name in your language-specific project directories. Even if you provide localized versions, however, you should always include a default version of these files at the top level of your application bundle. The default version is used in situations where a specific localization is not available.

At runtime, you can access your application’s resource files from your code using the following steps:

1. Obtain a reference to an appropriate bundle object (typically an `NSBundle` object).
2. Use the methods of that object to obtain a file-system path to the desired resource file.
3. Open or access the file using the standard system techniques.

You can obtain a reference to your application's main bundle using the `mainBundle` class method of `NSBundle`. The `pathForResource:ofType:` method is one of several methods that you can use to retrieve the location of resources in a bundle. The following example shows you how to locate a file called `sun.png` and create an image object using it. The first line gets the bundle object and the path to the file. The second line creates the `UIImage` object you would need in order to use the image in your application.

```
NSString* imagePath = [[NSBundle mainBundle] pathForResource:@"sun"
ofType:@"png"];
UIImage* sunImage = [[UIImage alloc] initWithContentsOfFile:imagePath];
```

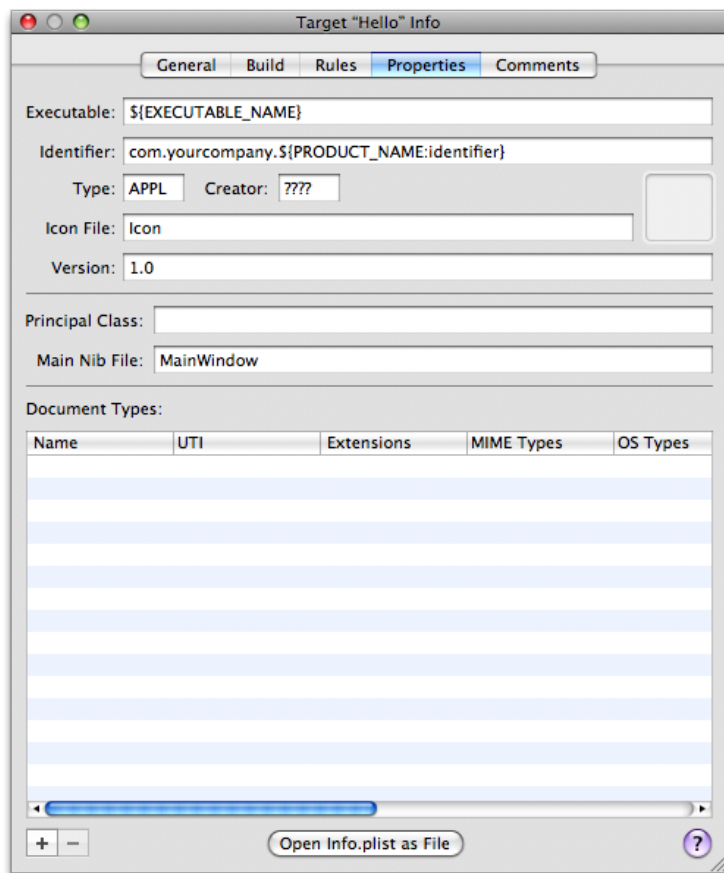
Note: If you prefer to use Core Foundation to access bundles, you can obtain a `CFBundleRef` opaque type using the `CFBundleGetMainBundle` function. You can then use that opaque type plus the Core Foundation functions to locate any bundle resources.

For information on how to access and use resources in your application, see *Resource Programming Guide*. For more information about the structure of an iOS application bundle, see *Bundle Programming Guide*.

The Information Property List

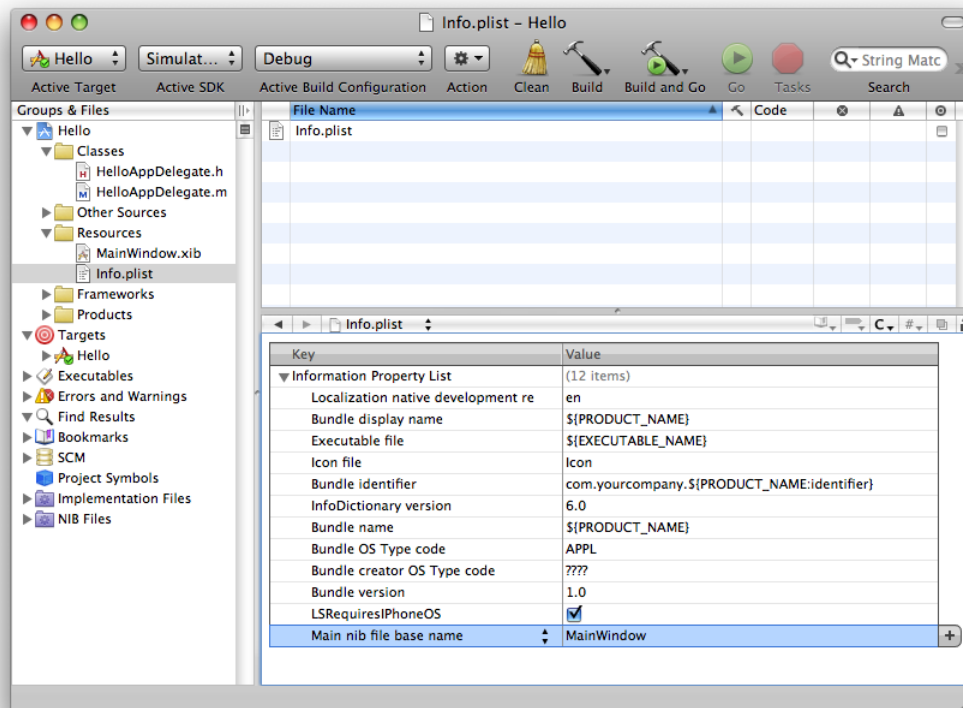
The information property list is a file named `Info.plist` that is included with every iOS application project created by Xcode. It is a property list whose key-value pairs specify essential runtime-configuration information for the application. The elements of the information property list are organized in a hierarchy in which each node is an entity such as an array, dictionary, string, or other scalar type.

In Xcode, you can access the information property list by choosing Edit Active Target *TargetName* from the Project menu. Then, in the target's Info window, click the Properties control. Xcode displays a pane of information similar to the example in Figure 7-1.

Figure 7-1 The Properties pane of a target's Info window

The Properties pane shows you some, but not all, of the properties of the application bundle. When you click the “Open Info.plist as File” button, or when you select the `Info.plist` file in your Xcode project, Xcode displays a property list editor window similar to the one in Figure 7-2. You can use this window to edit the property values and add new key-value pairs. To see the actual key names added to the `Info.plist` file, Control-click the Information Property List item in the editor and select Show Raw Keys/Values from the contextual menu that appears.

Figure 7-2 The information property list editor



Xcode automatically adds some important keys to the `Info.plist` file of all new projects and sets the initial value. However, there are several keys that iOS applications use commonly to configure their launch environment and runtime behavior. Here are some of the important keys that you might want to add to your application's `Info.plist` file specifically for iOS:

- `CFBundleIconFiles`
- `UIStatusBarStyle`
- `UIInterfaceOrientation`
- `UIRequiredDeviceCapabilities`
- `UIRequiresPersistentWiFi`

For detailed information about these and other properties that you can include in your application's `Info.plist` file, see *Information Property List Key Reference*.

iTunes Requirements

The App Store requires that you provide metadata about your application before submitting it. Most of this metadata is specified using the program portal web page, but some of it must be embedded directly in your application bundle.

Declaring the Required Device Capabilities

If your application requires device-related features in order to run, you must add a list of required capabilities to your application. At runtime, iOS will not launch your application unless those capabilities are present on the device. Further, the App Store requires this information so that it can generate a list of requirements for user devices and prevent users from downloading applications that they cannot run.

You add the list of required capabilities by adding the `UIRequiredDeviceCapabilities` key to your application's `Info.plist` file. This key is supported in iOS 3.0 and later. The value of this key is either an array or a dictionary. If you use an array, the presence of a given key indicates that the corresponding feature is required. If you use a dictionary, you must specify a Boolean value for each key. If the value of this key is `true`, the feature is required. If the value of the key is `false`, the feature must not be present on the device. In both cases, omitting a key indicates that the feature is not required but that the application is able to run if the feature is present.

Table 7-2 lists the keys that you can include in the array or dictionary associated with the `UIRequiredDeviceCapabilities` key. You should include keys only for the features that your application absolutely requires. If your application can accommodate missing features by not executing the appropriate code paths, you do not need to include the corresponding key.

Table 7-2 Dictionary keys for the `UIRequiredDeviceCapabilities` key

Key	Description
<code>telephony</code>	Include this key if your application requires (or specifically prohibits) the presence of the Phone application. You might require this feature if your application opens URLs with the <code>tel</code> scheme.
<code>wifi</code>	Include this key if your application requires (or specifically prohibits) access to the networking features of the device.
<code>sms</code>	Include this key if your application requires (or specifically prohibits) the presence of the Messages application. You might require this feature if your application opens URLs with the <code>sms</code> scheme.
<code>still-camera</code>	Include this key if your application requires (or specifically prohibits) the presence of a camera on the device. Applications use the <code>UIImagePickerController</code> interface to capture images from the device's still camera.
<code>auto-focus-camera</code>	Include this key if your application requires (or specifically prohibits) auto-focus capabilities in the device's still camera. Although most developers should not need to include this key, you might include it if your application supports macro photography or requires sharper images in order to do some sort of image processing.
<code>front-facing-camera</code>	Include this key if your application requires (or specifically prohibits) the presence of a forward-facing camera. Applications use the <code>UIImagePickerController</code> interface to capture video from the device's camera.
<code>camera-flash</code>	Include this key if your application requires (or specifically prohibits) the presence of a camera flash for taking pictures or shooting video. Applications use the <code>UIImagePickerController</code> interface to control the enabling of this feature.

Key	Description
video-camera	Include this key if your application requires (or specifically prohibits) the presence of a camera with video capabilities on the device. Applications use the <code>UIImagePickerController</code> interface to capture video from the device's camera.
accelerometer	Include this key if your application requires (or specifically prohibits) the presence of accelerometers on the device. Applications use the classes of the Core Motion framework to receive accelerometer events. You do not need to include this key if your application detects only device orientation changes.
gyroscope	Include this key if your application requires (or specifically prohibits) the presence of a gyroscope on the device. Applications use the Core Motion framework to retrieve information from gyroscope hardware.
location-services	Include this key if your application requires (or specifically prohibits) the ability to retrieve the device's current location using the Core Location framework. (This key refers to the general location services feature. If you specifically need GPS-level accuracy, you should also include the <code>gps</code> key.)
gps	Include this key if your application requires (or specifically prohibits) the presence of GPS (or AGPS) hardware when tracking locations. (You should include this key only if you need the higher accuracy offered by GPS hardware.) If you include this key, you should also include the <code>location-services</code> key. You should require GPS only if your application needs more accurate location data than the cell or Wi-fi radios might otherwise allow.
magnetometer	Include this key if your application requires (or specifically prohibits) the presence of magnetometer hardware. Applications use this hardware to receive heading-related events through the Core Location framework.
gamekit	Include this key if your application requires (or specifically prohibits) Game Center (iOS 4.1 and later.)
microphone	Include this key if your application uses the built-in microphone or supports accessories that provide a microphone.
opengles-1	Include this key if your application requires (or specifically prohibits) the presence of the OpenGL ES 1.1 interfaces.
opengles-2	Include this key if your application requires (or specifically prohibits) the presence of the OpenGL ES 2.0 interfaces.
armv6	Include this key if your application is compiled only for the armv6 instruction set. (iOS 3.1 and later.)
armv7	Include this key if your application is compiled only for the armv7 instruction set. (iOS 3.1 and later.)
peer-peer	Include this key if your application requires (or specifically prohibits) peer-to-peer connectivity over Bluetooth. (iOS 3.1 and later.)

For detailed information on how to create and edit property lists, see *Information Property List Key Reference*.

Application Icons

Every application must specify an icon to be displayed on the device's Home screen and in the App Store. And an application may specify several different icons for use in different situations. For example, applications can provide a small version of the application icon to use when displaying search results.

To specify the icons for your application, add the `CFBundleIconFiles` key to your application's `Info.plist` file and list the icon image filenames in the associated array. The filenames can be anything you want but all image files must be in the PNG format and reside in the top-level of your application bundle. When the system needs an appropriately sized icon, it looks at the files in the `CFBundleIconFiles` array and picks the one whose size most closely matches the intended usage. (If your application also runs in iOS 3.1.3 or earlier, you must use specific names for your icon image files; these filenames are described later in this section.)

Table 7-3 lists the dimensions of the image files you can associate with the `CFBundleIconFiles` key, along with basic information about the intended usage for each icon. For devices that support high-resolution versions of an icon, two icons should be provided with the second one being a high-resolution version of the original. The names of the two icons should be the same except for the inclusion of the string `@2x` in the filename of the high-resolution image as described in [“Updating Your Image Resource Files”](#) (page 75). For detailed information about the usage and preparation of your icons, see *iPhone Human Interface Guidelines* or *iPad Human Interface Guidelines*.

Table 7-3 Sizes for images in the `CFBundleIconFiles` key

Icon	Idiom	Size	Usage
Application icon (required)	iPhone	57 x 57 pixels 114 x 114 pixels (@2x)	This is the main icon for applications running on iPhone and iPod touch. The <code>@2x</code> variant of the icon is for use on iPhone 4 devices only.
Application icon (required)	iPad	72 x 72 pixels	This is the main icon for applications running on iPad.
Settings/Search results icon	iPhone/iPad	29 x 29 pixels 58 x 58 pixels (@2x)	Displayed in conjunction with search results on iPhone and iPod touch. Also used by the Settings application on all device types. (The <code>@2x</code> variant of the icon is for use on iPhone 4 devices only and is not supported on iPad.)
Search results icon	iPad	50 x 50 pixels	Displayed in conjunction with search results on iPad.

When specifying icon files using the `CFBundleIconFiles` key, it is best to omit the filename extensions of your image files. If you include a filename extension, you must explicitly add all of your image files (including any high-resolution variants) to the array. When you omit the filename extension, the system automatically detects high-resolution variants of your file using the base filename you provide.

Note: Do not confuse the `CFBundleIconFiles` key with the `CFBundleIconFile` key. The keys provide similar behaviors but the plural version is preferred because it allows you to specify an array of image filenames instead of a single filename. The plural version of the key is available only in iOS 3.2 and later.

If your iPhone application supports running in iOS 3.1.3 or earlier, you must use specific naming conventions when creating your icon image files. Because the `CFBundleIconFiles` key was introduced in iOS 3.2, it is not recognized by earlier versions of the system. Instead, earlier versions of the system look for application icons by name. The sizes of the icons are the same as those listed in [Table 7-3](#) (page 100), but each icon must be named as follows:

- `Icon.png` - The name for the application icon on iPhone or iPod touch.
- `Icon-72.png` - The name for the application icon on iPad.
- `Icon-Small.png` - The name for the search results icon on iPhone and iPod touch. This file is also used for the Settings icon on all devices.
- `Icon-Small-50.png` - The name of the search results icon on iPad.

Important: The use of fixed filenames for your application icons is for compatibility with earlier versions of iOS only. Even if you use these fixed icon filenames, your application should continue to include the `CFBundleIconFiles` key in its `Info.plist` if it is able to run in iOS 3.2 and later. In iOS 3.2 and earlier, the system looks for icons with the fixed filenames first. In iOS 4 and later, the system looks for icons in the `CFBundleIconFiles` key first.

In addition to the other icon files, developers who distribute their applications using ad hoc distribution must include a 512 x 512 version of their icon and give it the name `iTunesArtwork` (no filename extension). This icon is displayed by iTunes when presenting your application for distribution. Like all other icon files, the `iTunesArtwork` image file must reside at the top-level of your application bundle. The file should be the same one you would have submitted to the App Store (typically a JPEG or PNG file), were you to distribute your application that way.

For more information about the `CFBundleIconFiles` key, see *Information Property List Key Reference*. For information about creating your application icons, see *iPhone Human Interface Guidelines* and *iPad Human Interface Guidelines*.

Application Launch Images

When it launches an application, the system temporarily displays a static launch image on the screen. The launch image is an image file that you provide and is usually a prerendered version of your application's default user interface. Presenting this image gives the user immediate feedback that the application launched and also gives your application time to initialize itself and prepare its initial set of views for display. Once your application is ready to run, the image is removed and the corresponding windows and views are shown.

Every application must provide at least one launch image. This is typically a file named `Default.png` that displays your application's initial screen in a portrait orientation. However, you can also provide other launch images to be used under different launch conditions. Launch images must be PNG files and must reside in the top level of your application's bundle directory. The name of each launch image indicates when it is to be used, and the basic format for launch image filenames is as follows:

`<basename><usage_specific_modifiers><scale_modifier><device_modifier>.png`

The `<basename>` portion of the filename is either the string `Default` or the custom string associated with the `UILaunchImageFile` key in your application's `Info.plist` file. The `<scale_modifier>` portion is the optional string `@2x` and should be included only on images intended for use on high-resolution screens. Other optional modifiers may also be included in the name, and several standard modifiers are discussed in the sections that follow.

Table 7-4 lists the typical dimensions for launch images in iOS applications. For all dimensions, the image width is listed first, followed by the image height. For precise information about which size launch image to use, and how to prepare your launch images, see *iPhone Human Interface Guidelines* or *iPad Human Interface Guidelines*.

Table 7-4 Typical launch image dimensions

Device	Portrait	Landscape
iPhone and iPod touch	320 x 480 pixels 640 x 960 pixels (high resolution)	Not supported
iPad	768 x 1004 pixels	1024 x 748 pixels

To demonstrate the naming conventions, suppose your iOS application's `Info.plist` file included the `UILaunchImageFile` key with the value `MyLaunchImage`. The standard resolution version of the launch image would be named `MyLaunchImage.png` and would be in a portrait orientation (480 x 320). The high-resolution version of the same launch image would be named `MyLaunchImage@2x.png`. If you did not specify a custom launch image name these files would need to be named `Default.png` and `Default@2x.png` respectively.

Providing Launch Images for Different Orientations

In iOS 3.2 and later, an iPad application can provide different launch images depending on whether the device is in a portrait or landscape configuration. To specify a launch image for a different orientation, you must add a special modifier string to the base file name of your standard launch image. Thus, the format of each file name becomes the following:

`<basename><orientation_modifier><scale_modifier><device_modifier>.png`

Table 7-5 lists the possible modifiers you can specify for the `<orientation_modifier>` value in your image filenames. As with any launch image, the file must be in the PNG format. These modifiers are supported for launch images used in iPad applications only; they are not supported for applications running on iPhone or iPod touch devices.

Table 7-5 Launch image orientation modifiers

Modifier	Description
<code>-PortraitUpsideDown</code>	Specifies an upside-down portrait version of the launch image. A file with this modifier takes precedence over a file with the <code>-Portrait</code> modifier for this specific orientation.

Modifier	Description
-LandscapeLeft	Specifies a left-oriented landscape version of the launch image. A file with this modifier takes precedence over a file with the -Landscape modifier for this specific orientation.
-LandscapeRight	Specifies a right-oriented landscape version of the launch image. A file with this modifier takes precedence over a file with the -Landscape modifier for this specific orientation.
-Portrait	Specifies the generic portrait version of the launch image. This image is used for right side-up portrait orientations and takes precedence over the Default.png image file (or your custom-named replacement for that file). If a file with the -PortraitUpsideDown modifier is not specified, this file is also used for upside-down portrait orientations as well.
-Landscape	Specifies the generic landscape version of the launch image. If a file with the -LandscapeLeft or -LandscapeRight modifier is not specified, this image is used instead. This image takes precedence over the Default.png image file (or your custom-named replacement for that file).
None	If you provide a launch image file with no orientation modifier, that file is used when no other orientation-specific launch image is available. In iOS 3.1.x and earlier, this is the only supported launch image file type and must be named Default.png.

For example, if you specify the value `MyLaunchImage` in the `UILaunchImageFile` key, the custom landscape and portrait launch images for your application would be named `MyLaunchImage-Landscape.png` and `MyLaunchImage-Portrait.png`. (The high-resolution variants of your images would be `MyLaunchImage-Landscape@2x.png` and `MyLaunchImage-Portrait@2x.png`.) If you did not specify a custom launch image filename, you would use the names `Default-Landscape.png` and `Default-Portrait.png`.

No matter which launch image is selected by the system, your application always launches in a portrait orientation initially and then rotates as needed to the correct orientation. Therefore, your `application:didFinishLaunchingWithOptions:` method should always assume a portrait orientation when setting up your window and views. Shortly after your `application:didFinishLaunchingWithOptions:` method returns, the system sends any necessary orientation-change notifications to your application's window, giving it and your application's view controllers a chance to reorient any views using the standard process. For more information about how your view controllers manage the rotation process, see “Custom View Controllers” in *View Controller Programming Guide for iOS*.

Providing Device-Specific Launch Images

It is possible to mark specific image files as usable only on a specific type of device. This capability simplifies the code you have to write for Universal applications. Rather than creating separate code paths to load one version of a resource file for iPhone and a different version of the file for iPad, you can let the bundle-loading routines choose the correct file at load time. All you have to do is specify one of the following values for the `<device_modifier>` portion of your image's filename:

- `~ipad` - The resource should be loaded on iPad devices only.
- `~iphone` - The resource should be loaded on iPhone or iPod touch devices only.

For example, if the name of your application's launch image on iPhone and iPod touch is `MyLaunchImage.png`, the iPad specific version would be `MyLaunchImage~ipad.png`. Although you could include a `MyLaunchImage~iphone.png` image for the iPhone specific version (and a `MyLaunchImage@2x~iphone.png` for the high-resolution variant), it is better to use device modifiers only when necessary to distinguish the image from the default version. If no device-specific version of an image is present, the system loads the default version automatically.

Note: Another way to specify device-specific launch images is using device-specific keys in your `Info.plist` file. Adding a device-specific modifier to the `UILaunchImageFile` key lets you specify a custom basename for launch images used on different devices. For more information on how to apply device modifiers to keys in the `Info.plist` file, see *Information Property List Key Reference*.

Providing Launch Images for Custom URL Schemes

If your application supports a custom URL scheme, the system can display a custom launch image whenever it launches your application in response to a URL request. Prior to opening your application, the system looks for a launch image whose filename matches the following format:

`<basename>-<url_scheme><scale_modifier><device_modifier>.png`

The `<url_scheme>` modifier is a string representing the name of your URL scheme name. For example, if your application supports a URL scheme with the name `myscheme`, then the system would look for an image with the name `Default-myscheme.png` (or `Default-myscheme@2x.png` in the high-resolution case) in the application's bundle. If the application's `Info.plist` file includes the `UILaunchImageFile` key, then the base name portion changes from `Default` to the custom string you provide in that key.

Note: You can combine a URL scheme modifier with orientation modifiers. If you do this, the format for the file name is `<basename>-<scheme>-<orientation_modifier><scale_modifier><device_modifier>.png`. For more information about the launch orientation modifiers, see [“Providing Launch Images for Different Orientations”](#) (page 102).

In addition to including the launch images at the top level of your bundle, you can also include localized versions of that images in your application's language-specific project subdirectories. For more information on localizing resources in your application, see [“Internationalizing Your Application”](#) (page 104).

Internationalizing Your Application

Ideally, the text, images, and other content that iOS applications display to users should be localized for multiple languages. The text that an alert dialog displays, for example, should be in the preferred language of the user. The process of preparing a project for content localized for particular languages is called internationalization. Project components that are candidates for localization include:

- Code-generated text, including locale-specific aspects of date, time, and number formatting

- Static text—for example, an HTML file loaded into a web view for displaying application help
- Icons (including your application icon) and other images when those images either contain text or have some culture-specific meaning
- Sound files containing spoken language
- Nib files

Using the Settings application, users select the language they want to see in their applications' user interfaces from the Language preferences view (see Figure 7-3). They get to this view from the International group of settings, accessed from General settings.

Figure 7-3 The Language preference view



The chosen language is associated with a subdirectory of the bundle. The subdirectory name has two components: an ISO 639-1 language code and a `.lproj` suffix. For example, to designate resources localized to English, the bundle would be named `en.lproj`. By convention, these language-localized subdirectories are called `lproj` directories.

Note: You may use ISO 639-2 language codes instead of those defined by ISO 639-1. However, you should not include region codes (as defined by the ISO 3166-1 conventions) when naming your `lproj` directories. Although region information is used for formatting dates, numbers, and other types of information, it is not taken into consideration when choosing which language directory to use. For more information about language and region codes, see “Language and Locale Designations” in *Internationalization Programming Topics*.

An `lproj` directory contains all the localized content for a given language. You use the facilities of the `NSBundle` class or the `CFBundleRef` opaque type to locate (in one of the application's `lproj` directories) resources localized for the currently selected language. Listing 7-1 gives an example of such a directory containing content localized for English (`en`).

Listing 7-1 The contents of a language-localized subdirectory

```
en.lproj/  
    InfoPlist.strings  
    Localizable.strings  
    sign.png
```

This subdirectory example has the following items:

- The `InfoPlist.strings` file contains strings assigned as localized values of certain properties in the project's `Info.plist` file (such as `CFBundleDisplayName`). For example, the `CFBundleDisplayName` key for an application named `Battleship` in the English version would have this entry in `InfoPlist.strings` in the `fr.lproj` subdirectory:

```
CFBundleDisplayName = "Cuirassé";
```

- The `Localizable.strings` file contains localized versions of strings generated by application code.
- The `sign.png` file in this example is a file containing a localized image.

To internationalize strings in your code for localization, use the `NSLocalizedString` macro in place of the string. This macro has the following declaration:

```
NSString *NSLocalizedString(NSString *key, NSString *comment);
```

The first parameter is a unique key to a localized string in a `Localizable.strings` file in a given `lproj` directory. The second parameter is a comment that indicates how the string is used and therefore provides additional context to the translator. For example, suppose you are setting the content of a label (`UILabel` object) in your user interface. The following code would internationalize the label's text:

```
label.text = NSLocalizedString(@"City", @"Label for City text field");
```

You can then create a `Localizable.strings` file for a given language and add it to the proper `lproj` directory. For the above key, this file would have an entry similar to the following:

```
"City" = "Ville";
```

Note: Alternatively, you can insert `NSLocalizedString` calls in your code where appropriate and then run the `genstrings` command-line tool. This tool generates a `Localizable.strings` template that includes the key and comment for each string requiring translation. For further information about `genstrings`, see the `genstrings(1)` man page.

For more information about internationalization and how you support it in your iOS applications, see *Internationalization Programming Topics*.

Tuning for Performance and Responsiveness

At each step in the development of your application, you should consider the implications of your design choices on the overall performance of your application. The operating environment for iOS applications is more constrained because of the mobile nature of iPhone and iPod touch devices. The following sections describe the factors you should consider throughout the development process.

Do Not Block the Main Thread

You should be careful what work you perform from the main thread of your application. The main thread is where your application handles touch events and other user input. To ensure that your application is always responsive to the user, you should never use the main thread to perform long-running tasks or to perform tasks with a potentially unbounded end, such as tasks that access the network. Instead, you should always move those tasks onto background threads. The preferred way to do so is to wrap each task in an operation object and add it to an operation queue, but you can also create explicit threads yourself.

Moving tasks into the background leaves your main thread free to continue processing user input, which is especially important when your application is starting up or quitting. During these times, your application is expected to respond to events in a timely manner. If your application's main thread is blocked at launch time, the system could kill the application before it even finishes launching. If the main thread is blocked at quitting time, the system could kill the application before it has a chance to write out crucial user data.

For more information about using operation objects and threads, see *Concurrency Programming Guide*.

Use Memory Efficiently

Because the iOS virtual memory model does not include disk swap space, applications are somewhat more limited in the amount of memory they have available for use. Using large amounts of memory can seriously degrade system performance and potentially cause the system to terminate your application. When you design, therefore, make it a high priority to reduce the amount of memory used by your application.

There is a direct correlation between the amount of free memory available and the relative performance of your application. Less free memory means that the system is more likely to have trouble fulfilling future memory requests. If that happens, the system can always remove code pages and other nonvolatile resources from memory. However, removing those resources may be only a temporary fix, especially when those resources are needed again a short time later. Instead, minimize your memory use in the first place, and clean up the memory you do use in a timely manner.

The following sections provide more guidance on how to use memory efficiently and how to respond when there is only a small amount of available memory.

Reduce Your Application's Memory Footprint

Table 8-1 lists some tips on how to reduce your application's overall memory footprint. Starting off with a low footprint gives you more room for the data you need to manipulate.

Table 8-1 Tips for reducing your application's memory footprint

Tip	Actions to take
Eliminate memory leaks.	Because memory is a critical resource in iOS, your application should not have any memory leaks. Allowing leaks to exist means your application may not have the memory it needs later. You can use the Instruments application to track down leaks in your code, both in the simulator and on actual devices. For more information on using Instruments, see <i>Instruments User Guide</i> .
Make resource files as small as possible.	Files reside on the disk but must be loaded into memory before they can be used. Property list files and images are two resource types where you can save space with some very simple actions. To reduce the space used by property list files, write those files out in a binary format using the <code>NSPropertyListSerialization</code> class. For images, compress all image files to make them as small as possible. (To compress PNG images—the preferred image format for iOS applications—use the <code>pngcrush</code> tool.)
Use Core Data or SQLite for large data sets.	If your application manipulates large amounts of structured data, store it in a Core Data persistent store or in a SQLite database instead of in a flat file. Both Core Data and SQLite provides efficient ways to manage large data sets without requiring the entire set to be in memory all at once. The Core Data feature was introduced in iOS 3.0.
Load resources lazily.	You should never load a resource file until it is actually needed. Prefetching resource files may seem like a way to save time, but this practice actually slows down your application right away. In addition, if you end up not using the resource, loading it simply wastes memory.
Build your program using Thumb.	Adding the <code>-mthumb</code> compiler flag can reduce the size of your code by up to 35%. However, if your application contains floating-point-intensive code modules and you are building your application for arm6, you should disable Thumb. If you are building your code for arm7, you should leave Thumb enabled.

Allocate Memory Wisely

iOS applications use a managed memory model, whereby you must explicitly retain and release objects. Table 8-2 lists tips for allocating memory inside your program.

Table 8-2 Tips for allocating memory

Tip	Actions to take
Reduce your use of autoreleased objects.	Objects released using the <code>autorelease</code> method stay in memory until you explicitly drain the autorelease pool or until the next time around your event loop. Whenever possible, avoid using the <code>autorelease</code> method when you can instead use the <code>release</code> method to reclaim the memory occupied by the object immediately. If you must create moderate numbers of autoreleased objects, create a local autorelease pool and drain it periodically to reclaim the memory for those objects before the next event loop.
Impose size limits on resources.	Avoid loading large resource files when a smaller one will do. Instead of using a high-resolution image, use one that is appropriately sized for iOS-based devices. If you must use large resource files, find ways to load only the portion of the file that you need at any given time. For example, rather than load the entire file into memory, use the <code>mmap</code> and <code>munmap</code> functions to map portions of the file into and out of memory. For more information about mapping files into memory, see <i>File-System Performance Guidelines</i> .
Avoid unbounded problem sets.	Unbounded problem sets might require an arbitrarily large amount of data to compute. If the set requires more memory than is available, your application may be unable to complete the calculations. Your applications should avoid such sets whenever possible and work on problems with known memory limits.

For detailed information on how to allocate memory in iOS applications, and for more information on autorelease pools, see “Cocoa Objects” in *Cocoa Fundamentals Guide*.

Floating-Point Math Considerations

The processors found in iOS-based devices are capable of performing floating-point calculations in hardware. If you have an existing program that performs calculations using a software-based fixed-point math library, you should consider modifying your code to use floating-point math instead. Hardware-based floating-point computations are typically much faster than their software-based fixed-point equivalents.

Important: Of course, if you build your application for arm6 and your code does use floating-point math extensively, remember to compile that code without the `-mthumb` compiler option. The Thumb option can reduce the size of code modules, but it can also degrade the performance of floating-point code. If you build your application for arm7, you should always enable the Thumb option.

In iOS 4 and later, you can also use the functions of the Accelerate framework to perform complex mathematical calculations. This framework contains high-performance vector-accelerated libraries for digital signal processing and linear algebra mathematics. You can apply these libraries to problems involving audio and video processing, physics, statistics, cryptography, and complex algebraic equations.

Reduce Power Consumption

Power consumption on mobile devices is always an issue. The power management system in iOS conserves power by shutting down any hardware features that are not currently being used. In addition to avoiding CPU-intensive operations or operations that involve high graphics frame rates, you can help improve battery life by optimizing your use of the following features:

- The CPU
- Wi-Fi, Bluetooth, and baseband (EDGE, 3G) radios
- The Core Location framework
- The accelerometers
- The disk

The goal of your optimizations should be to do the most work you can in the most efficient way possible. You should always optimize your application's algorithms using Instruments. But it is important to remember that even the most optimized algorithm can still have a negative impact on a device's battery life. You should therefore consider the following guidelines when writing your code:

- Avoid doing work that requires polling. Polling prevents the CPU from going to sleep. Instead of polling, use the `NSRunLoop` or `NSTimer` classes to schedule work as needed.
- Leave the `idleTimerDisabled` property of the shared `UIApplication` object set to `NO` whenever possible. The idle timer turns off the device's screen after a specified period of inactivity. If your application does not need the screen to stay on, let the system turn it off. If your application experiences side effects as a result of the screen turning off, you should modify your code to eliminate the side effects rather than disable the idle timer unnecessarily.
- Coalesce work whenever possible, to maximize idle time. It takes more power to do work periodically over an extended period of time than it does to perform the same amount of work all at once. Performing work periodically prevents the system from powering down hardware over a longer period of time.
- Avoid over-accessing the disk. For example, if your application saves state information to the disk, do so only when that state information changes and coalesce changes whenever possible to avoid writing small changes at frequent intervals.
- Do not draw to the screen faster than needed. Drawing is an expensive operation when it comes to power. Do not rely on the hardware to throttle your frame rates. Draw only as many frames as your application actually needs.
- If you use the `UIAccelerometer` class to receive regular accelerometer events, disable the delivery of those events when you do not need them. Similarly, set the frequency of event delivery to the smallest value that is suitable for your needs. For more information, see *Event Handling Guide for iOS*.

The more data you transmit to the network, the more power must be used to run the radios. In fact, accessing the network is the most power-hungry operation you can perform and should be minimized by following these guidelines:

- Connect to external network servers only when needed, and do not poll those servers.
- When you must connect to the network, transmit the smallest amount of data needed to do the job. Use compact data formats and do not include excess content that will simply be ignored.

- Transmit data in bursts rather than spreading out transmission packets over time. The system turns off the Wi-Fi and cell radios when it detects a lack of activity. When it transmits data over a longer period of time, your application uses much more power than when it transmits the same amount of data in a shorter amount of time.
- Connect to the network using the Wi-Fi radios whenever possible. Wi-Fi uses less power and is preferred over the baseband radios.
- If you use the Core Location framework to gather location data, disable location updates as soon as you can and set the distance filter and accuracy levels to appropriate values. Core Location uses the available GPS, cell, and Wi-Fi networks to determine the user's location. Although Core Location works hard to minimize the use of these radios, setting the accuracy and filter values gives Core Location the option to turn off hardware altogether in situations where it is not needed. For more information, see *Location Awareness Programming Guide*.

The Instruments application includes several instruments for gathering power-related information. You can use these instruments to gather general information about power consumption and to gather specific measurements for hardware such as the Wi-Fi and Bluetooth radios, GPS receiver, display, and CPU. For more information about using these instruments, see *Instruments User Guide*.

Tune Your Code

iOS comes with several applications for tuning the performance of your application. Most of these tools run on Mac OS X and are suitable for tuning some aspects of your code while it runs in the simulator. For example, you can use the simulator to eliminate memory leaks and make sure your overall memory usage is as low as possible. You can also remove any computational hotspots in your code that might be caused by an inefficient algorithm or a previously unknown bottleneck.

After you have tuned your code in the simulator, you should then use the Instruments application to further tune your code on a device. Running your code on an actual device is the only way to tune your code fully. Because the simulator runs in Mac OS X, it has the advantage of a faster CPU and more usable memory, so its performance is generally much better than the performance on an actual device. And using Instruments to trace your code on an actual device may point out additional performance bottlenecks that need tuning.

For more information on using Instruments, see *Instruments User Guide*.

Improve File Access Times

When creating files or writing out file data, keep the following guidelines in mind:

- Minimize the amount of data you write to the disk. File operations are relatively slow and involve writing to the Flash disk, which has a limited lifespan. Some specific tips to help you minimize file-related operations include:
 - Write only the portions of the file that changed, but aggregate changes when you can. Avoid writing out the entire file just to change a few bytes.
 - When defining your file format, group frequently modified content together so as to minimize the overall number of blocks that need to be written to disk each time.

- ❑ If your data consists of structured content that is randomly accessed, store it in a Core Data persistent store or a SQLite database. This is especially important if the amount of data you are manipulating could grow to be more than a few megabytes in size.
- Avoid writing cache files to disk. The only exception to this rule is when your application quits and you need to write state information that can be used to put your application back into the same state when it is next launched.

Tune Your Networking Code

The networking stack in iOS includes several interfaces over the radio hardware of iPhone and iPod touch devices. The main programming interface is the `CFNetwork` framework, which builds on top of BSD sockets and opaque types in the Core Foundation framework to communicate with network entities. You can also use the `NSStream` classes in the Foundation framework and the low-level BSD sockets found in the Core OS layer of the system.

The following sections provide iOS-specific tips for developers who need to incorporate networking features into their applications. For information about how to use the `CFNetwork` framework for network communication, see *CFNetwork Programming Guide* and *CFNetwork Framework Reference*. For information about using the `NSStream` class, see *Foundation Framework Reference*.

Tips for Efficient Networking

When implementing code to receive or transmit across the network, remember that doing so is one of the most power-intensive operations on a device. Minimizing the amount of time spent transmitting or receiving helps improve battery life. To that end, you should consider the following tips when writing your network-related code:

- For protocols you control, define your data formats to be as compact as possible.
- Avoid communicating using chatty protocols.
- Transmit data packets in bursts whenever you can.

The cellular and Wi-Fi radios are designed to power down when there is no activity. Doing so can take several seconds though, depending on the radio. If your application transmits small bursts of data every few seconds, the radios may stay powered up and continue to consume power, even when they are not actually doing anything. Rather than transmit small amounts of data more often, it is better to transmit a larger amount of data once or at relatively large intervals.

When communicating over the network, it is also important to remember that packets can be lost at any time. When writing your networking code, you should be sure to make it as robust as possible when it comes to failure handling. It is perfectly reasonable to implement handlers that respond to changes in network conditions, but do not be surprised if those handlers are not called consistently. For example, the Bonjour networking callbacks may not always be called immediately in response to the disappearance of a network service. The Bonjour system service does immediately invoke browsing callbacks when it receives a notification that a service is going away, but network services can disappear without notification. This might occur if the device providing the network service unexpectedly loses network connectivity or the notification is lost in transit.

Using Wi-Fi

If your application accesses the network using the Wi-Fi radios, you must notify the system of that fact by including the `UIRequiresPersistentWiFi` key in the application's `Info.plist` file. The inclusion of this key lets the system know that it should display the network selection panel if it detects any active Wi-Fi hot spots. It also lets the system know that it should not attempt to shut down the Wi-Fi hardware while your application is running.

To prevent the Wi-Fi hardware from using too much power, iOS has a built-in timer that turns off the hardware completely after 30 minutes if no application has requested its use through the `UIRequiresPersistentWiFi` key. If the user launches an application that includes the key, iOS effectively disables the timer for the duration of the application's life cycle. As soon as that application quits, however, the system reenables the timer.

Note: Note that even when `UIRequiresPersistentWiFi` has a value of `true`, it has no effect when the device is idle (that is, screen-locked). The application is considered inactive, and although it may function on some levels, it has no Wi-Fi connection.

For more information on the `UIRequiresPersistentWiFi` key and the keys of the `Info.plist` file, see [“The Information Property List”](#) (page 95).

The Airplane Mode Alert

If the device is in airplane mode when your application launches, the system may display a panel to notify the user of that fact. The system displays this panel only when all of the following conditions are met:

- Your application's information property list (`Info.plist`) file contains the `UIRequiresPersistentWiFi` key and the value of that key is set to `true`.
- Your application launches while the device is currently in airplane mode.
- Wi-Fi on the device has not been manually reenabled after the switch to airplane mode.

Document Revision History

This table describes the changes to *iOS Application Programming Guide*.

Date	Notes
2010-08-20	Fixed several typographical errors and updated the code sample on initiating background tasks.
2010-06-30	Updated the guidance related to specifying application icons and launch images.
	Changed the title from <i>iPhone Application Programming Guide</i> .
2010-06-14	Reorganized the book so that it focuses on the design of the core parts of your application.
	Added information about how to support multitasking in iOS 4 and later. For more information, see “Multitasking” (page 36).
	Updated the section describing how to determine what hardware is available.
	Added information about how to support devices with high-resolution screens.
	Incorporated iPad-related information.
2010-02-24	Made minor corrections.
2010-01-20	Updated the “Multimedia Support” chapter with improved descriptions of audio formats and codecs.
2009-10-19	Moved the iPhone specific <code>Info.plist</code> keys to <i>Information Property List Key Reference</i> .
	Updated the “Multimedia Support” chapter for iOS 3.1.
2009-06-17	Added information about using the compass interfaces.
	Moved information about OpenGL support to <i>OpenGL ES Programming Guide for iOS</i> .
	Updated the list of supported <code>Info.plist</code> keys.
2009-05-14	Updated for iOS 3.0.
	Added code examples to “Copy and Paste Operations” in the Event Handling chapter.
	Added a section on keychain data to the Files and Networking chapter.
	Added information about how to display map and email interfaces.

Date	Notes
	Made various small corrections.
2009-01-06	Fixed several typos and clarified the creation process for child pages in the Settings application.
2008-11-12	Added guidance about floating-point math considerations
	Updated information related to what is backed up by iTunes.
2008-10-15	Reorganized the contents of the book.
	Moved the high-level iOS information to <i>iOS Technology Overview</i> .
	Moved information about the standard system URL schemes to <i>Apple URL Scheme Reference</i> .
	Moved information about the development tools and how to configure devices to <i>iOS Development Guide</i> .
	Created the Core Application chapter, which now introduces the application architecture and covers much of the guidance for creating iPhone applications.
	Added a Text and Web chapter to cover the use of text and web classes and the manipulation of the onscreen keyboard.
	Created a separate chapter for Files and Networking and moved existing information into it.
	Changed the title from <i>iOS Programming Guide</i> .
2008-07-08	New document that describes iOS and the development process for iPhone applications.