
iOS Application Programming Guide

General



2011-02-24



Apple Inc.
© 2011 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.

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.

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

App Store is a service mark of Apple Inc.

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

iPad and Retina are trademarks of Apple Inc.

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

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

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

UNIX is a registered trademark of The Open Group

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

Understanding the iOS Runtime Environment 10
Designing the Core of Your Application 10
Supporting Common Application Behaviors 10
Executing Code in the Background 10
Meeting the App Store and System Requirements 11
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 13
 The Automatic Sleep Timer 14
 Multitasking Support 14
Security 14
 The Application Sandbox 14
 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
iOS Simulator 19
Determining the Available Hardware Support 19

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 37
 Being a Responsible, Multitasking-Aware Application 37

Responding to System Changes While in the Background	39
Opting Out of Background Execution	41
Windows, Views, and View Controllers	41
The Event-Handling System	42
The Graphics and Drawing System	44
The Text System	44
Audio and Video Support	46
Integration with the Hardware and System Applications	46

Chapter 3 Implementing Common Application Behaviors 49

Preserving the State of Your Application's User Interface	49
Launching in Landscape Mode	50
Adding Support for AirPlay	50
Files and the File System	51
Getting Paths to Standard Application Directories	51
Sharing Files with the User	52
Working with Protected Files	53
Opening Files Whose Type Is Unknown	54
Implementing Support for Custom File Formats	56
Communicating with Other Applications	58
Implementing Custom URL Schemes	59
Registering Custom URL Schemes	59
Handling URL Requests	60
Displaying Application Preferences	62
Turning Off Screen Locking	63

Chapter 4 Executing Code in the Background 65

Determining Whether Multitasking Support Is Available	65
Declaring the Background Tasks You Support	65
Implementing Long-Running Background Tasks	66
Tracking the User's Location	66
Playing Background Audio	67
Implementing a VoIP Application	68
Completing a Finite-Length Task in the Background	70
Scheduling the Delivery of Local Notifications	71

Chapter 5 Implementing Application Preferences 73

The Settings Application Interface	73
The Settings Bundle	75
The Settings Page File Format	76
Hierarchical Preferences	76
Localized Resources	77
Creating and Modifying the Settings Bundle	78

- Adding the Settings Bundle 78
- Preparing the Settings Page for Editing 78
- Configuring a Settings Page: A Tutorial 79
- Creating Additional Settings Page Files 82
- Accessing Your Preferences 83
- Specifying Default Values for Preferences 83
- Debugging Preferences for Simulated Applications 84

Chapter 6 Build-Time Configuration Details 85

- The Application Bundle 85
- The Information Property List 87
- iTunes Requirements 90
 - Declaring the Required Device Capabilities 90
 - Application Icons 92
- Application Launch Images 94
 - Providing Launch Images for Different Orientations 95
 - Providing Device-Specific Launch Images 96
 - Providing Launch Images for Custom URL Schemes 96
- Creating a Universal Application 97
 - Configuring Your Xcode Project 97
 - Updating Your Info.plist Settings 98
 - Updating Your View Controllers and Views 99
 - Adding Runtime Checks for Newer Symbols 99
 - Using Runtime Checks to Create Conditional Code Paths 100
 - Updating Your Resource Files 100
- Using a Single Xcode Project to Build Two Applications 101
- Internationalizing Your Application 102

Chapter 7 Tuning for Performance and Responsiveness 105

- Do Not Block the Main Thread 105
- Use Memory Efficiently 105
 - Observing Low-Memory Warnings 106
 - Reduce Your Application's Memory Footprint 106
 - Allocate Memory Wisely 107
- Floating-Point Math Considerations 108
- Reduce Power Consumption 108
- Tune Your Code 110
- Improve File Access Times 110
- Tune Your Networking Code 110
 - Tips for Efficient Networking 111
 - Using Wi-Fi 111
 - The Airplane Mode Alert 112

Document Revision History 113

Figures, Tables, and Listings

Chapter 1 **The Application Runtime Environment 13**

Table 1-1	Directories of an iOS application	16
Table 1-2	Identifying available 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	31
Figure 2-4	Moving from the foreground to the background	32
Figure 2-5	Handling application interruptions	34
Figure 2-6	Transitioning from the background to the foreground	35
Figure 2-7	Processing events in the main run loop	43
Figure 2-8	Several different keyboards and input methods	45
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	39
Table 2-5	System integration technologies	46
Listing 2-1	The <code>main</code> function of an iOS application	28

Chapter 3 **Implementing Common Application Behaviors 49**

Figure 3-1	Defining a custom URL scheme in the <code>Info.plist</code> file	60
Table 3-1	Commonly used search path constants	51
Table 3-2	Keys and values of the <code>CFBundleURLTypes</code> property	59
Listing 3-1	Getting the path to the application's <code>Documents</code> directory	52
Listing 3-2	Document type information for a custom file format	57
Listing 3-3	Handling a URL request based on a custom scheme	61

Chapter 4 **Executing Code in the Background 65**

Table 4-1	Configuring stream interfaces for VoIP usage	69
Listing 4-1	Checking for background support in earlier versions of iOS	65
Listing 4-2	Starting a background task at quit time	71
Listing 4-3	Scheduling an alarm notification	72

Chapter 5 **Implementing Application Preferences 73**

Figure 5-1	Organizing preferences using child panes	77
------------	--	----

Figure 5-2	Formatted contents of the <code>Root.plist</code> file	79
Figure 5-3	A root Settings page	80
Table 5-1	Preference control types	74
Table 5-2	Contents of the <code>Settings.bundle</code> directory	75
Table 5-3	Root-level keys of a preferences Settings page file	76
Listing 5-1	Accessing preference values in an application	83

Chapter 6 **Build-Time Configuration Details 85**

Figure 6-1	The information property list editor	88
Figure 6-2	The Properties pane of a target's Info window	89
Figure 6-3	The Language preference view	103
Table 6-1	A typical application bundle	85
Table 6-2	Dictionary keys for the <code>UIRequiredDeviceCapabilities</code> key	90
Table 6-3	Sizes for images in the <code>CFBundleIconFiles</code> key	92
Table 6-4	Typical launch image dimensions	94
Table 6-5	Launch image orientation modifiers	95

Chapter 7 **Tuning for Performance and Responsiveness 105**

Table 7-1	Tips for reducing your application's memory footprint	106
Table 7-2	Tips for allocating memory	107

About iOS Application Design

This document is the starting point for learning how to create iOS applications. It contains fundamental information about the iOS environment and how your applications interact with that environment. It also contains important information about the architecture of iOS applications and tips for designing key parts of your application.



The contents of this document apply to all iOS applications running on all types of iOS devices, including iPad, iPhone, and iPod touch.

Note: Development of iOS applications requires an Intel-based Macintosh computer with the iOS SDK installed.

Understanding the iOS Runtime Environment

The iOS runtime environment was designed to support the needs of mobile users and mobile devices. This environment requires you to design applications differently than you might for a desktop operating system. By design, iOS places restrictions on applications to more effectively manage resources and the overall security of the system. These restrictions encompass everything from the way you manage memory and files in your application to how your application interacts with the device hardware.

Relevant Chapter: [“The Application Runtime Environment”](#) (page 13)

Designing the Core of Your Application

The UIKit framework provides the core for all iOS applications. This framework provides the infrastructure for creating and managing your user interface, handling events, supporting multitasking, and managing most other interactions with the system. Other system frameworks may provide interesting features but without the UIKit framework, your application would not run. Understanding the key objects of this framework (and the design patterns it uses) is therefore a critical part of understanding how to design your application.

Relevant Chapter: [“The Core Application Design”](#) (page 23)

Supporting Common Application Behaviors

There are several common behaviors that any iOS application might want to implement—for example, launching your application in a landscape orientation, registering a custom URL type, or doing interesting things with files. Most of these behaviors require modifying your core application slightly to support them.

Relevant Chapters: [“Implementing Common Application Behaviors”](#) (page 49)
[“Implementing Application Preferences”](#) (page 73)

Executing Code in the Background

Few applications should ever need to execute code while in the background. The basic multitasking support provides applications with the ability to remain in the background in a suspended state, thereby preserving battery life. However, applications that provide specific services to the user may ask the system for permission to run in the background so that they can continue providing those services.

Relevant Chapter: [“Executing Code in the Background”](#) (page 65)

Meeting the App Store and System Requirements

Your application’s information property list file (`Info.plist`) conveys important information about your application to the system and to the App Store. This file resides in your application bundle, which provides the fundamental structure for organizing your application’s resources and localized content. Configuring both your bundle and `Info.plist` file correctly is an important part of building any application.

Relevant Chapter: [“Build-Time Configuration Details”](#) (page 85)

Tuning Performance for the Underlying Device

In iOS, good performance is particularly important and can mean the difference between success and failure for an application. If your application is sluggish or interferes with other applications, users are less likely to buy it. And because resources such as memory are more constrained on iOS-based devices, it is imperative that you respond to memory warnings and address leaks.

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.

Relevant Chapter: [“Tuning for Performance and Responsiveness”](#) (page 105)

See Also

In addition to this document, the following documents provide fundamental information you need to design iOS applications:

- For information about the design patterns used by iOS frameworks, see *Cocoa Fundamentals Guide*.
- For information about user interface design and how to create effective applications using iOS, see *iOS Human Interface Guidelines*.
- For general information about all iOS technologies, see *iOS Technology Overview*.
- For a tutorial to get you started creating iOS applications, see *Your First iOS Application*.

There are many documents that provide detailed information about specific aspects of application design. You should refer to these documents when implementing specific portions of your application.

- For information about handling touch events and motion-related events, see *Event Handling Guide for iOS*.
- For information about structuring and managing your application's user interface, see *View Controller Programming Guide for iOS*.
- For information about presenting and animating your user interface, see *View Programming Guide for iOS*.
- For information about drawing custom content, see *Drawing and Printing Guide for iOS*.
- For information about handling text and web content and about managing the keyboard, see *Text, Web, and Editing Programming Guide for iOS*.
- For information about incorporating audio and video into your application, see *Multimedia Programming Guide*.

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 an iPhone or iPad out of a pocket or bag, uses it for a few seconds or minutes, and then puts it away. Your own applications need to reflect that sense of immediacy by launching and becoming ready to run quickly. If your application takes a long time to launch, the user may be less inclined to use it.

Even when multitasking is available, the user interacts with only one application at a time. Thus, as each application is launched, the interface for the previous application goes away. Before iOS 4, this meant that the previous application was quit and removed from memory. When multitasking is available, though, quitting an application puts it in the background, where it stays until it is launched again or removed from memory.

Multitasking makes the relaunching of applications much faster but does not eliminate the launching of applications altogether. As memory becomes constrained, the system purges applications that have not been used recently. Purges can happen at any time and with no notice. It is therefore imperative that applications save user data and any application state when they move to the background. During the next launch cycle, the application should then use that state information to restore the application to its previous state. Restoring the application in this way makes it seem as if the application never quit, which provides continuity and convenience for the user.

Specialized System Behaviors

The core iOS system is based on the same technologies used by Mac OS X, namely the Mach kernel and BSD interfaces. Thus, iOS applications run in a UNIX-based system and have full support for threads, sockets, and many of the other technologies typically available at that level. However, there are places where the behavior of iOS differs from that of 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, the amount of usable virtual memory is constrained by the amount of physical memory available. This is because iOS does not support

paging to disk when memory gets full. Instead, the virtual memory system simply releases read-only memory pages, such as code pages, when it needs more space. Such pages can always be loaded back into memory later if they are needed again.

If memory continues to be constrained, the system may send notifications to any 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 106).

The Automatic Sleep Timer

One way iOS saves battery power is through the automatic sleep timer. When the system does not detect touch events for an extended period of time, it dims the screen initially and eventually turns it off altogether.

If you are creating an application that does not use touch inputs, such as a game that relies on the accelerometers, you can disable the automatic sleep timer to prevent the screen from dimming. You should use this timer sparingly and reenable it as soon as possible to conserve power. Only applications that display visual content and do not rely on touch inputs should ever disable the timer. Audio applications or applications that do not need to present visual content should not disable the timer.

The process for disabling the timer is described in [“Turning Off Screen Locking”](#) (page 63). For additional tips on how to save power in your application, see [“Reduce Power Consumption”](#) (page 108).

Multitasking Support

In iOS 4 and later, multitasking allows applications to run in the background even when they are not visible on the screen. Most background applications reside in memory but do not actually execute any code. These applications are suspended by the system shortly after entering the background to preserve battery life. Applications can ask the system for background execution time in a number of ways, though.

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 protect 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 each 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. Each application has access to the contents of its own sandbox but cannot access other applications’ sandboxes.

When an application is first installed on a device, the system creates the application's home directory, sets up some key subdirectories, and sets up the security privileges for the sandbox. The path to home directory is of the following form:

/ApplicationRoot/ApplicationID/

The *ApplicationRoot* portion of the home directory path is the place in the file system where applications are stored. The *ApplicationID* portion is a unique and opaque identifier for the application itself. The use of a unique location for each application simplifies data storage options for the application, backup-and-restore operations, application updates, and uninstallation.

Inside an application's home directory, the system creates several standard directories for storing files. Applications may also create custom directories inside the home directory, or any of its subdirectories, as well. 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 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 cause your program to crash or use it to execute the attacker's code. You should therefore do everything you can to ensure that your application is protected against outside attacks.

File Protection

In iOS 4 and later, applications can use the file protection feature 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 and iPhone 4) 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 the following steps:

- The file system on the user's device must be formatted to support file protection. For existing devices, the user must 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 53).
- Applications must respond appropriately to situations where a file may become locked; see [“Determining the Availability of Protected Files”](#) (page 54).

It is your responsibility to choose which files to protect for your application. Applications must enable file protection on a file-by-file basis, and once enabled those protections cannot be removed. Applications should also be prepared to handle the case where data files are not currently protected but should be. This situation can happen if the user restores a device from an earlier backup when file protections had not yet been added.

For more information about implementing support for file protection in your application, see [“Working with Protected Files”](#) (page 53).

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. When the user backs up application data using iTunes, the keychain data is also backed up. Before iOS 4.0, keychain data could only be restored to the device from which the backup was made. In iOS 4.0 and later, a keychain item that is password protected can be restored to a different device only if its accessibility is not set to `kSecAttrAccessibleAlwaysThisDeviceOnly` or any other value that restricts it to the current device. Upgrading 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 of the device. 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.

For information about working with files in iOS, see *Low-Level File Management Programming Topics*.

A Few Important Application Directories

For security purposes, an application has limited options as to where 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 this 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 14).

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. Do not write anything to this directory. To prevent tampering, the bundle directory is signed at installation time. Writing to this directory changes the signature and prevents your application from launching again.</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>

Directory	Description
<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 files that are not user data files. You typically put files in one of several standard subdirectories but you can also create custom subdirectories for files you want backed up but not exposed to the user. (For information on how to get references to the standard subdirectories, see “Getting Paths to Standard Application Directories” (page 51).) 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 preference 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 78).</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 re-create these files as needed because iTunes removes them during a full restoration of the device.</p> <p>In iOS 2.2 and later, the contents of this directory are not backed up by iTunes.</p>
<code><Application_Home>/tmp/</code>	<p>Use this directory to write temporary files that 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 Standard Application Directories”](#) (page 51). 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, 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 between applications, use the pasteboard. To share files between applications, use a document interaction controller object.

For information on how to support file sharing in your application, see [“Sharing Files with the User”](#) (page 52).

Backup and Restore

The iTunes application automatically handles the backup and restoration of user data in appropriate situations. The location of files that your application creates determines whether or not those files are backed up and restored. As you write your file-management code, you need to be mindful of this fact.

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 a 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. Use the `<Application_Home>/Documents` directory to store user documents and application data files that need to be backed up. Files used to store temporary data should be placed in 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, 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. 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 16).

Files Saved During Application Updates

When a user downloads an 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 them being present after an update.

iOS Simulator

The iOS Simulator application is a tool you can use to test your applications before deploying them to the App Store. 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 Simulator. Also, technologies such as OpenGL ES may not behave the same way in Simulator as they would on a device.

For more information about 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 are always present, some devices may not include a camera or GPS hardware. If your application requires a feature, it should declare its support for that feature by including the `UIRequiredDeviceCapabilities` key in its `Info.plist` file. For features that are not required, but which you might want to support when they are present, check to see whether the feature is available before trying to use it.

Important: If a feature must be present in order for your application to run, you must include the `UIRequiredDeviceCapabilities` key in your application's `Info.plist` file. This key prevents users from installing applications on a device that does not support the feature. 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 90).

If your application can function with or without a specific feature, use runtime checks to see whether that feature is present before attempting to use it. Table 1-2 lists the techniques for determining whether certain types of hardware or features are available.

Table 1-2 Identifying available features

Feature	Options
To determine if multitasking is available...	<p>Get the value of the <code>multitaskingSupported</code> property in the <code>UIDevice</code> class.</p> <p>For more information about determining the availability of multitasking, see “Determining Whether Multitasking Support Is Available” (page 65).</p>
To determine if you should configure your interface for an iPad-sized screen or an iPhone-sized screen...	<p>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.</p> <p>For more information on implementing a universal application, see “Creating a Universal Application” (page 97).</p>
To determine if an external screen is attached...	<p>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.</p> <p>For more information about displaying content on an external display, see “Windows” in <i>View Programming Guide for iOS</i>.</p>
To determine if hardware-level disk encryption is available...	<p>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 53).</p>
To determine if the network is available...	<p>If you need to transfer something using the network, just do so and be prepared to handle any errors that might crop up. The <code>CFNetwork</code> framework (as well as <code>NSStream</code> and other network interfaces) report errors when they occur, so you should be sure to provide appropriate error handlers when using these interfaces.</p> <p>If you need specific information about the state of the network, you can use the reachability interfaces of the System Configuration framework to obtain that information. For an example of how to use the System Configuration framework, see the sample code project <i>Reachability</i>.</p>
To determine if the still camera is available...	<p>Use the <code>isSourceTypeAvailable:</code> method of the <code>UIImagePickerController</code> class to determine if a camera is available.</p> <p>For more information, see <i>Camera Programming Topics for iOS</i>.</p>

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>Camera Programming Topics for iOS</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...	Configure a <code>CLLocationManager</code> object for location updates and specify a high level of accuracy. 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 values reported in subsequent location events are 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 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, it needs to be notified so that it can save any relevant data before moving to the background. 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 same 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 the data displayed in that interface. The controller portion acts as a bridge between the model and view and coordinates the passing of data 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 blocks often 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.

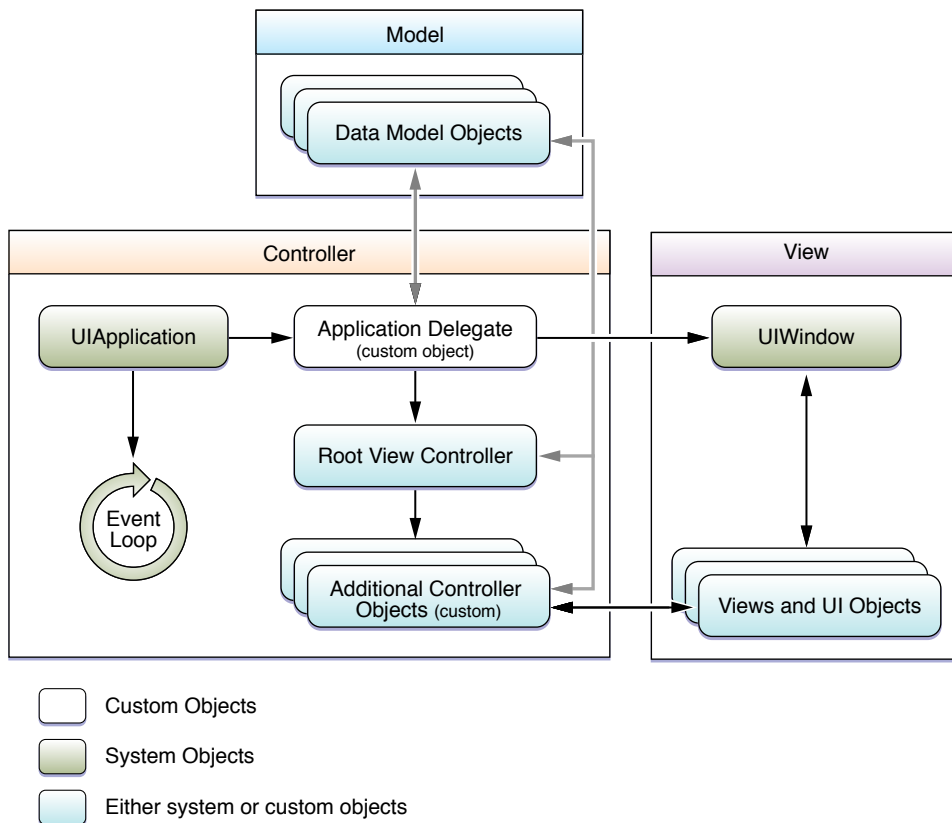
Design pattern	Description
Target-action	Controls use the target-action design pattern to notify your application of user interactions. When the user interacts with a control in a predefined way (such as by touching a button), the control sends a message (the action) to an object you specify (the target). Upon receiving the action message, the target object can then respond in an appropriate manner.
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 or 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) to execute 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 core behavior. For example, 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. Similar relationships exist in other parts of your application, with the system object managing the overall process and your custom code focused on implementing your application's specific behavior.

To understand how UIKit objects work with your custom code, it helps to understand a little about the objects that are involved. 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
UIApplication 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 moved to the background (because the user tapped the Home button). For more information about this object, see “The Application Delegate” (page 29).

Object	Description
Data model objects	<p>Data model objects store your application's content and are therefore specific to your application. For example, a banking application might store a database containing financial transactions, whereas 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.)</p>
View controller objects	<p>View controller objects manage the presentation of your application's content. Typically, this involves creating the views to present that content and managing the interactions between the views and 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 animating the appearance of views, handling device rotations, and many other standard system behaviors. UIKit and other frameworks also define other view controller classes for managing standard system interfaces, such as navigation interfaces or the image picker.</p> <p>For detailed information about how to use view controllers, see <i>View Controller Programming Guide for iOS</i>.</p>
UIWindow object	<p>A <code>UIWindow</code> object coordinates the presentation of one or more views on the device screen or on an external display. Most applications have only one window, the content of which is provided by one or more views. An application changes the content of that window by changing the current set of views (usually with the help of 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>
View, control, and layer objects	<p>Views and controls provide the visual representation of your application's content. A view is an object that draws 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>

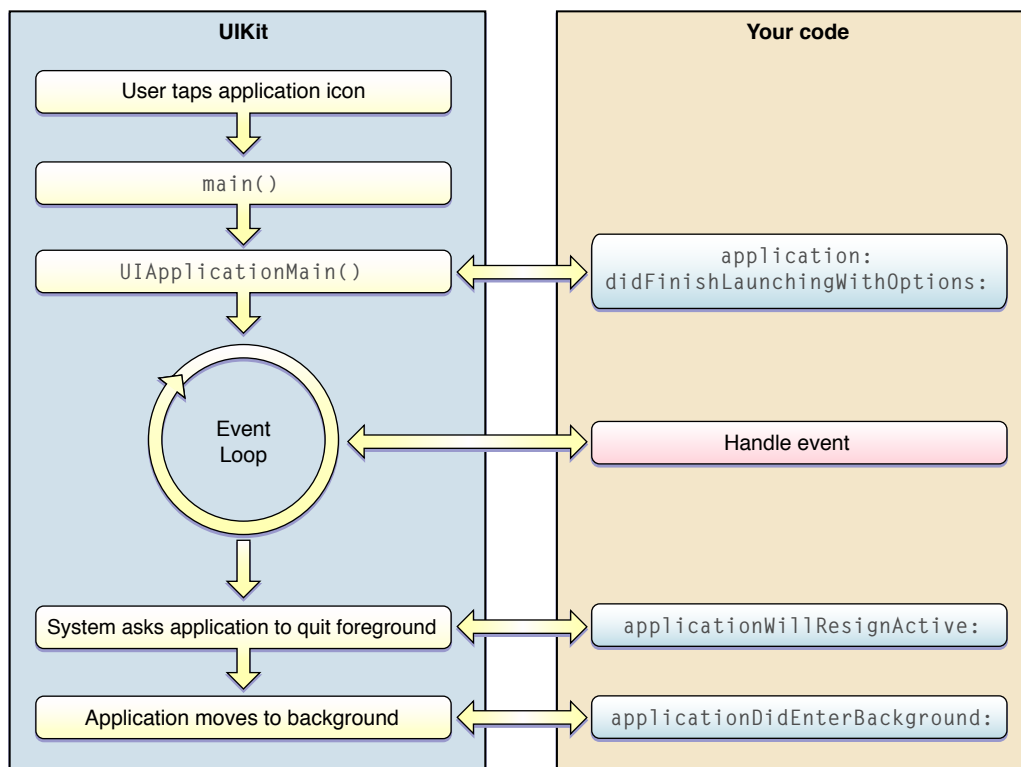
The objects in your application form a complex environment, the specifics of which are what define your application. As you can see from [Figure 2-1](#) (page 25), most of the objects in an application are either partially or wholly customizable. Fortunately, an application that builds on top of existing 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 and implement the customizations you need. The remainder of this chapter focuses on the places where you need to override the default behaviors to implement your application. It also points you to additional documents where you can find out more about specific types of system 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 main nib file and readies the event loop.

Figure 2-2 depicts the simplified startup life cycle for a newly launched iOS application. This diagram shows the sequence of events that occur between the time the application starts up and the point at which another application is launched. 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 also dispatches events to your application's custom event handlers, which are your views and view controllers.

Figure 2-2 Application life cycle



Before iOS 4, when the user quit an application, that application was terminated and removed from memory. This resulted in a simpler application model but at the cost of longer launch times. Keeping applications in memory adds complexity to the application life cycle but provides significant benefits to both the application and users.

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 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 107).

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 from the `NSMainNibFile` key in the 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 87).

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 `UIApplication` object. Instead of subclassing and overriding methods in a complex object, you use that object unmodified and put your custom code inside a 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, which are listed in Table 2-3. For applications running in iOS 3.2 and earlier, applications do not enter the background or suspended states.

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 in this state only briefly 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.

State	Description
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 65).</p> <p>The background state is available only 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>
Suspended	<p>The application is in the background but is not executing code. The system moves an 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 may purge suspended applications without notice to make more space for the foreground application.</p> <p>The suspended state is available only 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>

To facilitate movement between these transitions, the system calls the following methods of your application delegate.

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

The following sections describe the key state transitions that applications now experience, highlight the delegate methods that are called for each transition, and discuss the behaviors your application should adopt when making the transition.

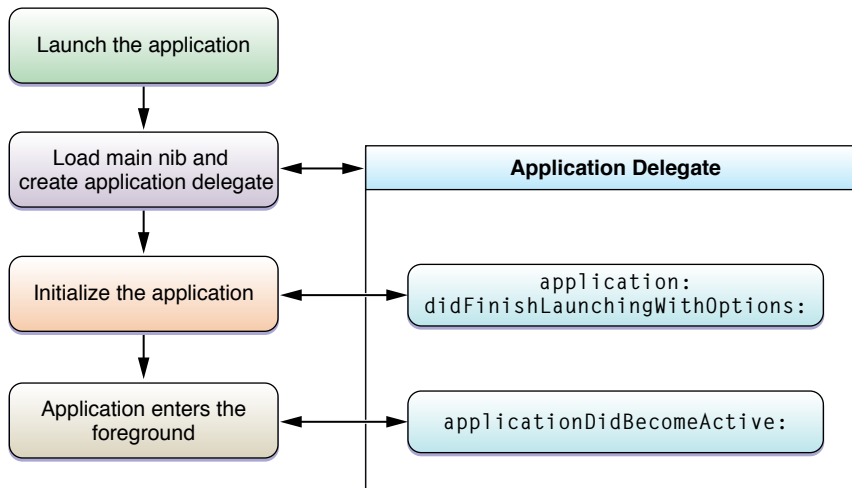
Launching the Application

At launch time, an application moves from the not-running state to the active or background state. A launching application has to prepare itself to run and then check to see whether 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 versions of iOS earlier than 4.0 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 main window and views in a portrait orientation.

If the device is in an orientation other than portrait at launch time, you should still create your interface in a portrait orientation initially. After the `application:didFinishLaunchingWithOptions:` method returns, the application corrects the orientation as needed by telling the window to rotate its contents. The window then uses the normal orientation-change mechanism of its view controllers to make the rotation happen before 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, it has only around 5 seconds to do its initialization and return. If it takes too long to complete its tasks, the system might kill the application for being unresponsive. Ways to make this method lightweight include initiating tasks asynchronously or moving any long-running tasks to secondary threads. Making the method lightweight is especially important for network-based tasks that could take an indeterminate amount of time to complete.

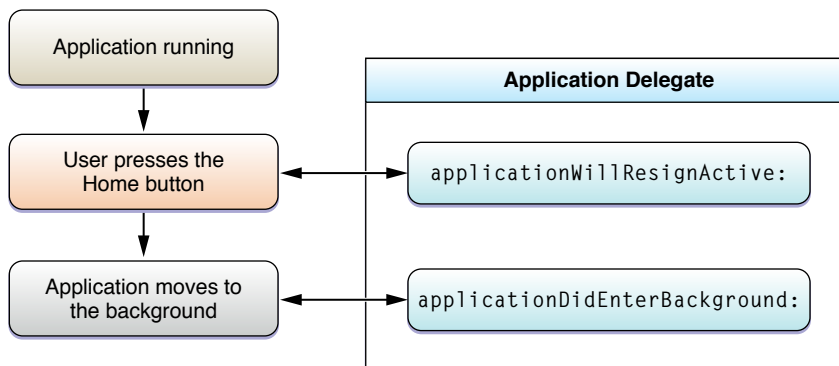
When your `application:didFinishLaunchingWithOptions:` method is called, the `applicationState` property of the `UIApplication` object is already set to the appropriate state for your application. If the property is set to `UIApplicationStateInactive`, your application is in the inactive state and is 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 appropriately.

Note: 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, 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 37).

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. These actions result 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 an application moves to the background, all of your core application objects remain in memory and are available for use. These objects include your custom objects and data structures plus your application's controller objects, windows, views, and layers. 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 does not release the layer objects themselves.
- 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.

Your delegate's `applicationDidEnterBackground:` method has approximately 5 seconds to finish any tasks and return. In practice, this method should return as quickly as possible. If the method does not return before time runs out, your application is killed and purged from memory. If you still need more time to perform tasks, call the `beginBackgroundTaskWithExpirationHandler:` method to request background execution time and then start any long-running tasks in a secondary thread. Regardless of whether you start any background tasks, the `applicationDidEnterBackground:` method must still exit within 5 seconds.

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

All multitasking-aware applications should behave responsibly when moving to the background. This is true regardless of whether your application continues running in the background or is suspended shortly after entering the background. To that end, there are at least two things you should always do in your `applicationDidEnterBackground:` method:

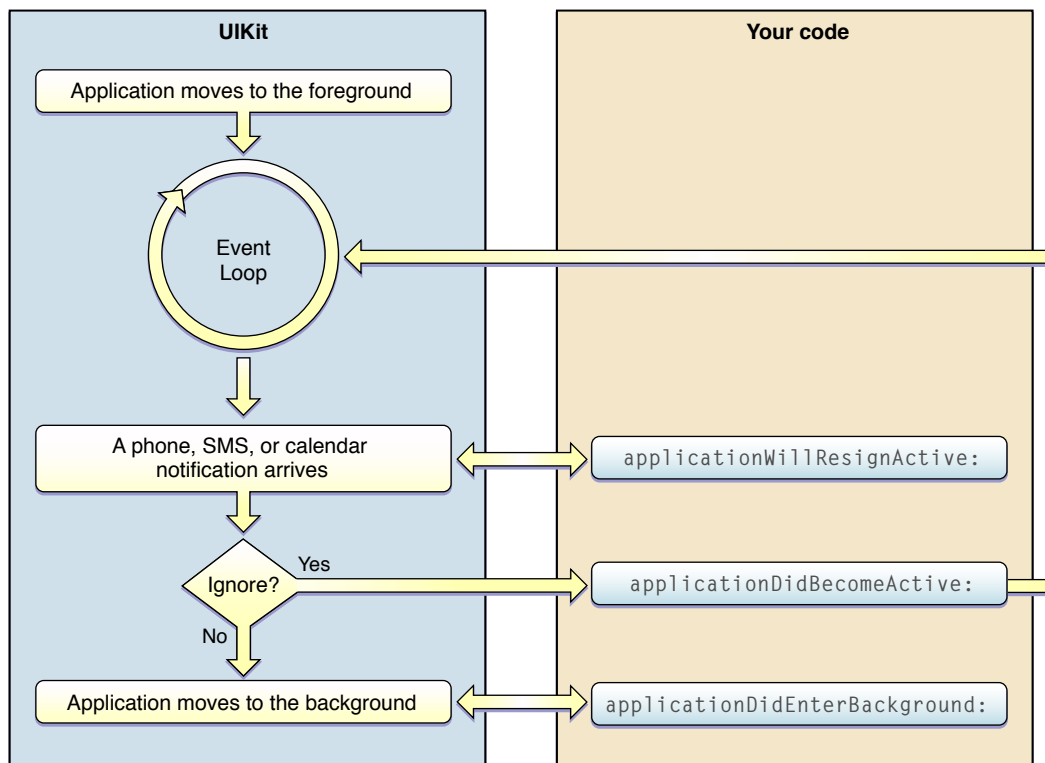
- **Prepare your application to have its picture taken.** When the `applicationDidEnterBackground:` method returns, the system takes a picture of your application's user interface and uses the resulting image for transition animations. If any views in your interface contain sensitive information, you should hide or modify those views before the `applicationDidEnterBackground:` method returns.
- **Save user data and application state information.** All unsaved changes should be written to disk when entering the background. This step is necessary because your application might be quietly killed while in the background for any number of reasons. You can perform this operation from a background thread as needed.

In addition to the preceding items, there are other tasks you might need to perform depending on your application's features. For example, applications that offer Bonjour services should end those services before being suspended. For a list of things all applications should do when moving to the background, see [“Being a Responsible, Multitasking-Aware Application”](#) (page 37).

If the application is running in a version of iOS earlier than 4.0 or is running 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 36).

Responding to Interruptions

When a temporary interruption (such as an incoming phone call) occurs, the application moves temporarily to the inactive state. It remains in this state until the user decides whether to accept or ignore the interruption. Upon moving to the inactive state, the application should put itself in a quiescent state as much as possible. If the user ignores the interruption, the application is reactivated, at which time it can resume its normal operation. However, if the user accepts the interruption, the application moves into the background state. Figure 2-5 shows the steps of this process. The text that follows describes the steps in more detail.

Figure 2-5 Handling application interruptions

1. An interruption such as an incoming phone call, SMS message, or 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 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 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 a version of iOS earlier than 4.0 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 relaunched 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.

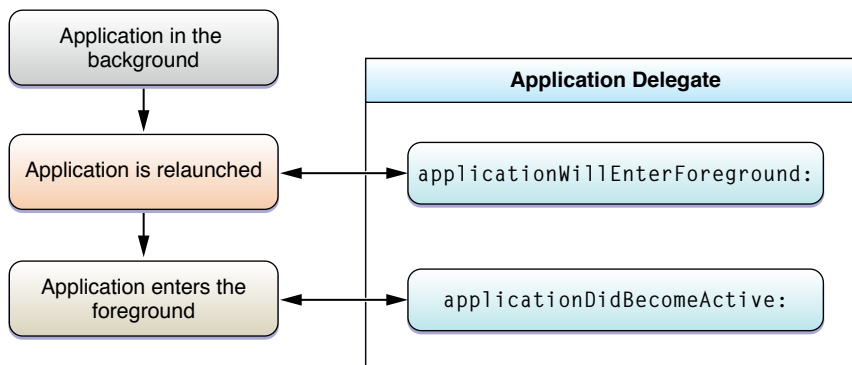
Important: When the user takes a call and then returns to your application while on the call, the height of the status bar grows to reflect the fact that the user is on a call. Similarly, when the user ends the call, the status bar height shrinks back to its regular size. Your application should be prepared 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 locks the screen. When the user unlocks the screen later, the system calls your application delegate's `applicationDidBecomeActive:` method and begins delivering events to the application again. While the screen is locked, foreground and background applications continue to run. However, when the screen locks, it is recommended that you put your own application into a quiescent state and do as little work as possible.

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. These actions result 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 in the suspended state, 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 39).

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 a version of iOS earlier than 4.0.
- The application is deployed on a device running a version of iOS earlier than 4.0.
- The current device does not support multitasking; see [“Determining Whether Multitasking Support Is Available”](#) (page 65).
- The application includes the `UIApplicationExitsOnSuspend` key in its `Info.plist` file; see [“Opting Out of Background Execution”](#) (page 41).

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 5 seconds to perform any tasks and return. If it does not return in time, the application is killed 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 killed without any notification. The user can kill applications explicitly using the multitasking UI. In addition, if memory becomes constrained, the system might remove applications from memory to make more room. If your application is currently suspended, the system kills your application and removes it 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 can 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 should do the following to support multitasking:

- **(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 it is in the background and in situations when it might need to be terminated. For information about these guidelines, see [“Being a Responsible, Multitasking-Aware Application”](#) (page 37).
- **(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 39).
- **(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 65).

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 41).

Being a Responsible, Multitasking-Aware Application

The foreground application always has precedence over other applications for the use of system resources and hardware. Therefore, applications need to adjust their behavior to handle transitions between the foreground and background. Specifically, applications moving to the background should follow these guidelines:

- **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 while running in the background. Using these calls causes your application to be killed 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 may be 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.
- **Release any unneeded memory when moving to the background.** If your application maintains a large in-memory cache of objects (especially images), you should consider releasing those caches when moving to the background. During low-memory conditions, the system considers the memory footprint of applications when deciding which ones to purge, with larger applications being the first to go. Releasing your caches helps alleviate memory pressure and makes it less likely that applications will need to be purged later.
- **Stop using shared system resources before being suspended.** Applications that interact with shared system resources such as the Address Book or calendar databases should stop using those resources before being suspended. Priority for such resources always goes to the foreground application. When your application is suspended, if it is found to be using a shared resource, the application is killed.
- **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 does not cause your application to be killed, 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. 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).

For applications linked against a version of iOS earlier than 4.0, action sheets and alerts are still dismissed at quit time so that your application's cancellation handler has a chance to run.

- **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 killed.

If you are implementing a background audio application, or any other type of application that is allowed to run 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 even more memory, the application calls its delegate's `applicationWillTerminate:` method to perform any final tasks before exiting.

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 receives 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 is updated automatically by UIKit. Your application needs to respond only if it tracks 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	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>

Event	Notification mechanism
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 changed	<code>NSUserDefaultsDidChangeNotification</code>
The current language or locale settings changed	<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 whether 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. You should also write your code in ways that make it easy to update views.

- 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 account information. Failure to monitor these changes could have serious privacy and security implications. Specifically, the current user might be able to send email using the old account information, even if the account no longer belongs 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 5 seconds to clean up and exit before it is terminated and moved back to the not-running state.

Even though opting out of background execution is strongly discouraged, it 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 kill 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 may have an additional window to support an external display. Each window fills its entire main screen and has no visual adornments such as a title bar or close box. A window is simply an empty surface that hosts one or more views.

A **view**, an instance of the `UIView` class, defines the content for a rectangular region of a window. Views are the primary mechanism for interacting with the user and have several responsibilities. For example:

- Drawing and animation support
 - Views draw content in their rectangular area.
 - Some view properties can animate to new values.
- Layout and subview management

- Each view manages a list of subviews, allowing you to create arbitrary view hierarchies.
 - Each view defines its own resizing behaviors (in relation to its parent view).
 - Each view can change the size and position of its subviews, either automatically or using custom code you provide.
- Event handling
 - Views receive touch events.
 - Views forward events to other objects when appropriate.

View controllers, instances of the `UIViewController` class, manage the structure of your user interface and the presentation of views within that interface. Applications have a limited amount of space in which to display content. Rather than display new windows, an application uses view controllers to swap out the views in the current window. The type of the view controller and the way that view controller is presented determine the type of animation that is used to display the new views.

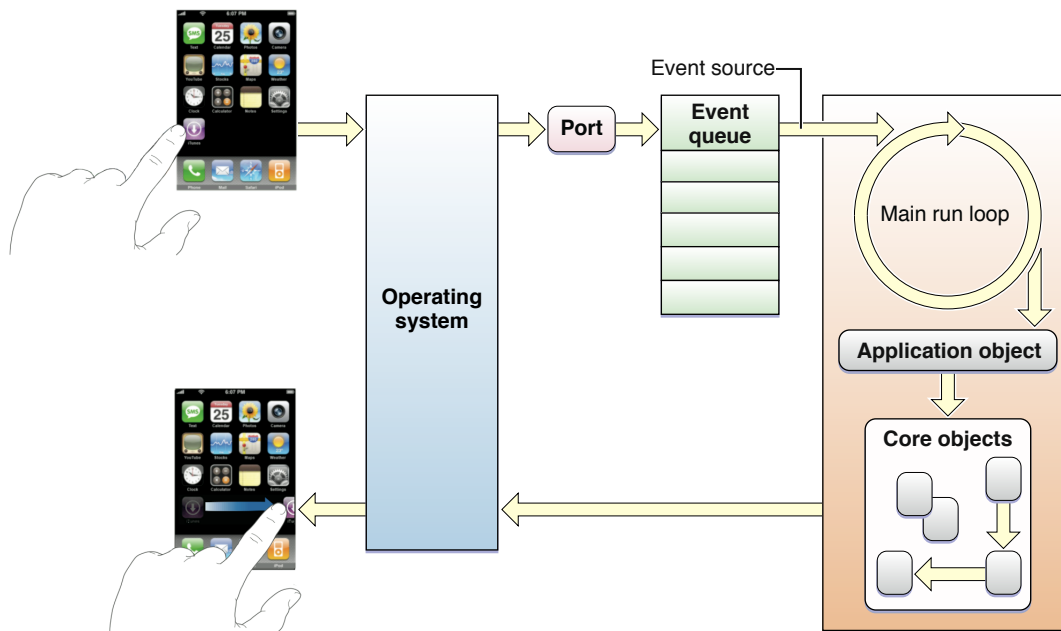
Each view controller object can be thought of as a self-contained unit. It handles the creation and destruction of its own views, handles their presentation on the screen, and coordinates interactions between the views and other objects in your application. The view controller may manage all of the views on its own or use helper objects to manage subsets of views. But the view controller is always in charge of presenting the overall view hierarchy and is ultimately responsible for the views.

Understanding view controllers and the infrastructure they provide is crucial 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 creating and configuring 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. Most 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. Touches are the most significant type of event you can receive, but other types of events may also be generated and delivered.

When the system launches an application, it creates both a process and a single thread for the application. This initial thread becomes the application's main thread. In it, 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 associated handler, which in the case of touch events is the `UIApplication` object. When the handler finishes, control passes back to the run loop, which can then process another event, process other input sources, or put 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*.

Most events sent to an application are encapsulated in an event object—an instance of the `UIEvent` class. 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 often sets 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.

Note: Accelerometer and gyro events are delivered using a different set of handlers that do not go through the standard responder chain.

For more information about events, responders, and handling events (including accelerometer and gyro events) in your own custom objects, see *Event Handling Guide for iOS*.

The Graphics and Drawing System

There are two basic ways an iOS application can draw its content:

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

The native iOS 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 your custom views must implement a `drawRect:` method. 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 update your view's visual content, you mark all or part of the view invalid by calling its `setNeedsDisplay` or `setNeedsDisplayInRect:` method. The system then calls your view's `drawRect:` method (at an appropriate time) to accommodate the update. 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 those objects simply provide the rendering surface for an OpenGL drawing context. Once you have that drawing context, your application is responsible for initiating drawing updates at appropriate intervals.

For information about drawing using the native iOS drawing technologies, see *Drawing and Printing Guide for iOS*. For 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

Applications that want to replace the keyboard can do so using a custom input view. Display of the keyboard is triggered by an object becoming the first responder. If you assign a custom input view to a responder object, that view is displayed instead. You can use input views to provide custom input options for your application's views. For example, a financial program might use an input view that is tailored for currency data entry.

Applications that need to display text have a variety of options. For simple text display and editing, you can use the `UILabel`, `UITextField`, and `UITextView` classes. You can also do simple string drawing using extensions 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 the keyboard, text input, and text drawing, see *Text, Web, and Editing Programming Guide for iOS*.

Audio and Video Support

For applications that use audio and video, iOS provides numerous technologies to support your needs. For video playback, use the Media Player and AV Foundation frameworks. For audio playback, you can use the same frameworks plus the Core Audio and OpenAL frameworks. Using these frameworks, you can implement features such as these:

- High-quality audio recording, playback, and streaming
- Immersive game sounds
- Live voice chat
- Playback of content from a user's iPod library
- Full- and partial-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 appear in the Calendar application and other applications that support this framework. For more information, see <i>Event Kit Programming Guide</i> .

Integration with...	Description
The Mail and Messages applications	<p>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.</p> <p>For more information, see <i>System Messaging Programming Topics for iOS</i>.</p>
Telephony information	<p>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.</p> <p>For more information, see <i>Core Telephony Framework Reference</i>.</p>
The camera hardware and the user's photo library	<p>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.</p> <p>For more information, see <i>Camera Programming Topics for iOS</i>.</p>
Unknown file types	<p>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.</p> <p>For information on how to use document interaction controllers, see “Opening Files Whose Type Is Unknown” (page 54).</p>
Pasteboard data	<p>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.</p> <p>For more information see <i>Text, Web, and Editing Programming Guide for iOS</i>.</p>
The location of the device	<p>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.</p> <p>For information about using Core Location, see <i>Location Awareness Programming Guide</i>.</p>
Map information	<p>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>.</p>
External hardware accessories	<p>Developers can create software that interacts with connected external hardware using the External Accessory framework.</p> <p>For information about communicating with external accessories, see <i>External Accessory Programming Topics</i>.</p>

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

Implementing Common Application Behaviors

Every application is different, but there are some application tasks that are the same for all applications. This chapter shows you how to implement some of the common tasks you might want to perform in iOS applications.

Preserving the State of Your Application's User Interface

An application can save the state of its user interface by walking its view controller hierarchy and saving information about each view controller to disk. Walking the view controllers is fast and enables you to gather enough information to restore your application to its previous state. As you walk your view controller hierarchy, you need to save the following information at a minimum:

- The currently visible view controller
- The structural arrangement of your view controllers
- Information about each view controller, including:
 - The class name of the view controller, which you use to recreate the view controller during the next launch cycle
 - References to the data being managed by the view controller

One approach to saving this information is to build a property list that is structured to match the organization of your view controllers. In this property list, you save information about each view controller in a dictionary object. The keys of the dictionary identify properties of the view controller, such as its class name and pointers to any relevant data objects. For container view controllers, such as navigation and tab bar controllers, the dictionary would also contain an array with the dictionaries for any child view controllers.

Practically speaking, your application should save information only about those view controllers that are not part of your application's default user interface. That is, when an application launches, it normally loads a main nib file or creates an initial set of views and view controllers. This initial set of view controllers provides the interface that users see when they first launch the application. Because these objects are always created, you may not need to save them in your property list.

When your application's `applicationDidEnterBackground:` or `applicationWillTerminate:` method is called, build your property list and save it as an application preference. Then, in your `application:didFinishLaunchingWithOptions:` method, load the property list from preferences and use it to create and configure any additional view controllers you need.

Launching in Landscape Mode

All iOS applications launch in portrait mode initially and rotate their interface to match the device orientation as needed. Thus, if your application supports both portrait and landscape orientations, you should always configure your views for portrait mode and then let your view controllers handle any rotations. If, however, your application supports landscape but not portrait orientations, you must perform the following tasks to make it appear to launch in landscape mode initially:

- Add the `UIInterfaceOrientation` key to your application's `Info.plist` file and set the value of this key to either `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` for the left or right landscape orientation and `NO` for portrait orientations.

Important: The preceding information assumes that 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 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 before 2.1, you need to apply a 90-degree rotation to the transform of the application's root view in addition to all the preceding steps.

Adding Support for AirPlay

If your application supports the streaming of audio or video using AirPlay, you should configure your application appropriately:

- Use the appropriate APIs to initiate the routing of audio and video content using AirPlay. For video, you must use the `MPMoviePlayerController` and `MPVolumeView` classes, whose interfaces include controls that allow the user to select a nearby AirPlay–enabled device and route content to it. For audio content, the system frameworks that manage audio playback (including the Media Player framework and the Core Audio family of frameworks) automatically handle AirPlay routing.

Note: For video, your application must explicitly opt-in to AirPlay video streaming before users can take advantage of it. Set the `allowsAirPlay` property of your `MPMoviePlayerController` object to `YES` before presenting your content to the user.

- If your application supports streaming content while in the background, include the `UIBackgroundModes` key (with the `audio` value) in your application's `Info.plist` file. When your application moves to the background, this key allows it to continue streaming content to an AirPlay-enabled device. Without it, your application would be suspended and streaming would stop. For more information about implementing a background audio application, see [“Playing Background Audio”](#) (page 67).
- Register and handle the media remote control events. These events let the user control your application when using the system transport controls or an external accessory, such as a headset, that supports the appropriate Apple specifications. For more information about supporting the media remote control events, see *“Remote Control of Multimedia”* in *Event Handling Guide for iOS*.

Files and the File System

Every application has its own designated area of the file system in which to store files. This area is part of the application sandbox and is accessible only to the application. Any operations that involve transferring files or content to other applications must go through well-defined and secure mechanisms provided by iOS.

Getting Paths to Standard Application Directories

Several frameworks provide programmatic ways to locate the standard directories in the application sandbox. However, the preferred way to retrieve these paths is using the `NSSearchPathForDirectoriesInDomains` and `NSTemporaryDirectory` functions. Both of these functions are part of the Foundation framework and are used to retrieve the locations of well-known directories. When you have the base path returned by one of these functions, you can use the path-related methods of the `NSString` class to modify the path information or create new path strings.

The `NSSearchPathForDirectoriesInDomains` function takes parameters indicating the directory you want to locate and the domain of that directory. For the domain, specify the `NSUserDomainMask` constant to restrict the search to your application. For the directory, the main constants you need are listed in Table 3-1.

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>

To get the location of your application's `<Application_Home>/tmp` directory, use the `NSTemporaryDirectory` function instead of the `NSSearchPathForDirectoriesInDomains` function.

Because the `NSSearchPathForDirectoriesInDomains` function was designed originally for Mac OS X, where there could be more than one of each of these directories, it returns an array of paths rather than a single path. In iOS, the resulting array should contain the single path to the 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 51), 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, the path `/Applications` is 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 calling `NSSearchPathForDirectoriesInDomains` on the device or in iOS Simulator. For example, take the function call shown in [Listing 3-1](#) (page 52). On the device, the returned path (`documentsDirectory`) is similar to the following:

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

However, in iOS Simulator, the returned path is of the following form:

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

To read and write user preferences, do not write to the file system directly; use the `NSUserDefaults` class or the `CFPreferences` API instead. 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 78).

If your application contains sound, image, or other resources in the application bundle, 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 85).

Sharing Files with the User

An application that wants to make files accessible to the user can do so through the file sharing feature. File sharing exposes the contents of an application's `Documents` directory in iTunes. Exposing this directory allows the user to add files to it or delete files from it. 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.

When the device is plugged into the user's computer, iTunes displays a File Sharing section in the Apps pane of the selected device. 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. You should never present users with the list of files in this directory and ask them to decide what to do with those files. Instead, sort through the files programmatically and add files without prompting. Similarly, if users deleted files, remove those files from any relevant parts of your user interface.

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. Take this into consideration when deciding where to put files in this directory.

If you do not want files to be shared with the user, put them in your application's `Library` directory. If none of the standard `Library` subdirectories are appropriate, create a private directory inside the `Library` directory and give it the name of your application's bundle ID. You can then use that directory to store any files that are private to your application.

File sharing does not allow you to share files directly with other applications on the same or a different device. To transfer files between applications, you must use a document interaction controller or the pasteboard. For information on how to use a document interaction controller, see [“Opening Files Whose Type Is Unknown”](#) (page 54).

Working with Protected Files

When file protection is enabled on a device, for greater security an application can mark files as protected. You might do this for files that contain private user data or sensitive information. If you plan to support this feature, though, your application must be prepared to handle situations where your application is running but 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 way guarantees that the data is always stored on disk in an encrypted format.

For new files, it is recommended that you add the `NSFileProtectionKey` attribute to the file before you write any data to it. (If you are writing out the contents of an `NSData` object, this protection happens automatically.) When adding this attribute to existing (unprotected) files, adding this key replaces the

unprotected file with a protected version. An application with write access to the protected file may change the `NSFileProtectionKey` attribute again later but may do so only while the file is writable—that is, only while the device is unlocked.

Determining the Availability of Protected Files

A protected file is accessible 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.

To use a document interaction controller, do the following:

1. Create an instance of the `UIDocumentInteractionController` class for each file you want to manage.
2. Create a view to represent the file in your application's user interface. (For example, use the view to display the filename or the icon.)
3. Attach the gesture recognizers from the document interaction controller to your view.
4. When the user touches the view, the gesture recognizers automatically trigger one of the following interfaces:
 - A tap gesture displays a preview of the file.
 - A long press gesture displays a menu with options to preview the file, copy its contents, or open it using another application.

Any application that manages files can use a document interaction controller. Programs that download files from the network are the most likely candidates, but any application can incorporate this behavior. For example, an email program might use document interaction controllers to preview or open files attached to an email.

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. For example, 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 are using a view to represent the file, attach the document interaction controller's gesture recognizers to your view to handle interactions automatically. The gesture recognizers in the `gestureRecognizers` property present appropriate interfaces whenever the user taps or does a long press gesture on your view. Using the gesture recognizers eliminates the need to create a custom view object just to handle touch events. Instead, you can use a standard `UIImageView` object to display the file's icon or any other type of standard system view.

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 can recognize. 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 made available to it.

Presenting a Document Interaction Controller

In addition to using the built-in gesture recognizers, you can initiate document interactions manually using the methods of the `UIDocumentInteractionController` class:

- Use the `presentOptionsMenuFromRect:inView:animated:` or `presentOptionsMenuFromBarButtonItem:animated:` method to present the user with options for previewing the file, copying its contents, or opening it in another application.

- 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 whether 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, the associated delegate object must implement the `documentInteractionControllerViewControllerForPreview:` 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:` method.

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:

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

Registering the File Types Your Application Supports

To register support for file types, your application must include the `CFBundleDocumentTypes` key in its `Info.plist` file. The system takes the information from this key and adds it to a private registry. When applications try to open unknown file types, the document interaction controller object uses this registry to determine which applications can open the file.

The `CFBundleDocumentTypes` key contains an array of dictionaries, each of which identifies information about a single document type. From the perspective of your application, a document type corresponds to 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.

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

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

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

Listing 3-2 shows a sample XML snippet from the `Info.plist` file 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

At any time, the system may ask your application to open a specific file and present it to the user. When the system makes the request, your application will either be launched or (if it is running in the background) be moved to the foreground. At that time, the system provides your application delegate with information about the file.

- Use the `application:didFinishLaunchingWithOptions:` method to retrieve information about the file at launch time and decide whether you want to open it.
- In iOS 4.2 and later, use the `application:openURL:sourceApplication:annotation:` method to open the file.
- In iOS 4.1 and earlier, use the `application:handleOpenURL:` method to open the file.

The `application:didFinishLaunchingWithOptions:` method is called only when your application must be launched to open a file. You typically do not open the file in this method, though. Instead, you return YES if your application is able to open the file. The system then calls the `application:openURL:sourceApplication:annotation:` or `application:handleOpenURL:` method to actually open the file.

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. You should use the information in these keys to determine whether you want to open the file. Specifically, this dictionary may contain the following keys:

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

If your application is already running when a document request arrives, the sequence of events depends on your delegate implementation and the current version of iOS. In iOS 4.2, the `application:openURL:sourceApplication:annotation:` method is called and is expected to check the document and open it. If your delegate does not implement that method, or if your application is running in iOS 4.1 and earlier, only the `application:handleOpenURL:` method is called.

Modifying Documents Passed to Your Application by the System

Before asking an application to open a file, the Mail application and `UIDocumentInteractionController` objects move the file to the `~/Documents/Inbox` directory of the target application. Thus, the application that opens the file receives the file locally in its sandbox. Only the system can place files in this directory. The application receiving the files can read them and delete them, but it cannot create files or write to them while they reside in the `Inbox` directory.

If your application allows the user to edit files located in the `~/Documents/Inbox` directory, copy those files out of the `Inbox` directory and into 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 delete the original file in the `Inbox` directory after the copy is made. The process of copying the file to a writable directory should be transparent to the user. Do not prompt the user for a new location to save the file. Have a designated location in mind and make the transfer behind the scenes.

Communicating with Other Applications

Applications that support custom URL schemes can use those schemes to receive messages. Some applications use URL schemes to initiate specific requests. For example, an application that wants to show an address in the Maps application can use a URL to launch that application and display the address. You can implement your own URL schemes to facilitate similar types of communications in your applications.

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. The handlers for these schemes are fixed and cannot be changed. 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.

Note: If more than one third-party application registers to handle the same URL scheme, there is currently no process for determining which application will be given that scheme.

To communicate with an application using a custom URL, create an `NSURL` object with some properly formatted content and pass that object to the `openURL:` method of the shared `UIApplication` object. The `openURL:` method launches the application that registered to receive URLs of that type and passes it the URL. At that point, control passes to the new application.

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];
```

If your application defines a custom URL scheme, it should implement a handler for that scheme as described in ["Implementing Custom URL Schemes"](#) (page 59). 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

If your application can receive specially formatted URLs, you should register the corresponding URL schemes with the system. A custom URL scheme is a mechanism through which third-party applications can communicate with each other. Applications often use custom URL schemes to vend services to other applications. For example, the Maps application supports URLs for displaying specific map locations.

Registering Custom URL Schemes

To register a URL type for your application, include the `CFBundleURLTypes` key in your application's `Info.plist` file. The `CFBundleURLTypes` key contains an array of dictionaries, each of which defines a URL scheme the application supports. Table 3-2 describes the keys and values to include in each dictionary.

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

Key	Value
<code>CFBundleURLName</code>	<p>A string containing the abstract name of the URL scheme. 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 string you specify is also used as a key in your application's <code>InfoPlist.strings</code> file. The value of the key is the human-readable scheme name.</p>

Key	Value
CFBundleURLSchemes	An array of strings containing the URL scheme names—for example, <code>http</code> , <code>mailto</code> , <code>tel</code> , and <code>sms</code> .

Figure 3-1 shows the `Info.plist` file of an application that supports a custom scheme for creating “to-do” items. The `URL types` entry corresponds to the `CFBundleURLTypes` key added to the `Info.plist` file. Similarly, the “URL identifier” and “URL Schemes” entries correspond 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 registering a custom URL scheme, 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 Simulator, you can go to the Home screen by choosing Hardware > Home from the menu.)
3. In the address field 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, either at launch time or while your application is running or in the background. To handle incoming URLs, your delegate should implement the following methods:

- 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.

- In iOS 4.2 and later, use the `application:openURL:sourceApplication:annotation:` method to open the file.
- In iOS 4.1 and earlier, use the `application:handleOpenURL:` method to open the file.

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. The implementation of your `application:didFinishLaunchingWithOptions:` method should retrieve the URL from its options dictionary and determine whether the application can open it. If it can, return `YES` and let your `application:openURL:sourceApplication:annotation:` or `application:handleOpenURL:` method handle the actual opening of the URL.

If your application is running but is in the background or suspended, it is moved to the foreground to open the URL. Shortly thereafter, the system calls the delegate's `application:openURL:sourceApplication:annotation:` to check the URL and open it. If your delegate does not implement this method (or the current system version is iOS 4.1 or earlier), the system calls your delegate's `application:handleOpenURL:` method instead.

Note: Applications that support custom URL schemes specify different launch images to be displayed when launching the application to handle a URL. For more information about how to specify these launch images, see [“Providing Launch Images for Custom URL Schemes”](#) (page 96).

All URLs are passed to your application in an `NSURL` object. It is up to you to define the format of the URL, but the `NSURL` class conforms to the RFC 1808 specification and therefore supports most URL formatting conventions. Specifically, the class 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. This example assumes that the user is using a Gregorian calendar. If your application supports non-Gregorian calendars, you need to design your URL scheme accordingly and be prepared to handle those other calendar types in your code.

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 alloc]
initWithCalendarIdentifier:NSGregorianCalendar]
                                autorelease];
        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 the user might change frequently should be presented from inside your application using a custom interface.

- Preferences that are not likely to change frequently should be handled using a Settings bundle.

A **Settings bundle** is a custom resource that you create using Xcode. A Settings bundle contains a specification for how the Settings application should display your preferences. Specifically, it indicates what controls and labels should be displayed and how values entered by the user should be saved to the preferences database. Settings bundles can also contain custom images for controls that support them.

Note: If you implement 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 92).

Regardless of which option you choose, you access preferences using the `NSUserDefaults` or `CFPreferences` APIs. These interfaces let you query the preferences database for the value associated with a specific key. You also use them to set keys or provide a set of default values to use in places where the user has not yet provided a value.

For more information about creating a Settings bundle for your application, see [“Implementing Application Preferences”](#) (page 73). For more information about retrieving preferences in your application, see *User Defaults Programming Topics*.

Turning Off Screen Locking

If an iOS-based device does not receive touch events for a specified period of time, the system turns off the screen and disables the touch sensor. Locking the screen is an important way to save power. As a result, you should generally leave this feature enabled. However, an application that does not rely on touch events, such as a game that uses the accelerometers for input, can disable screen locking to prevent the screen from going dark while the application is running. However, even in this case, screen locking should be disabled only while the user is actively engaged with the application. For example, if the user pauses a game, you should reenable screen locking to allow the screen to turn off.

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.

Executing Code in the Background

Most applications that enter the background state are moved to the suspended state shortly thereafter. While in this state, the application does not execute any code and may be removed from memory at any time. Applications that provide specific services to the user can request background execution time in order to provide those services.

Important: Most application developers do not need to read this chapter. This chapter does not address the basic multitasking support that all applications should adopt. It addresses only the steps needed to support executing code while the application is in the background state. For information about how to support basic multitasking in your applications, see [“Multitasking”](#) (page 36).

Applications linked against iOS SDK 4 and later are automatically assumed to support basic multitasking and to implement the appropriate methods to handle transitions to the background state.

Determining Whether Multitasking Support Is Available

The ability to execute code in the background is not supported on all iOS-based devices. Even devices running iOS 4 or later may not have the hardware to support multitasking. In those cases, the system reverts to the previously defined behavior for handling applications. Specifically, when an application quits, it is terminated and purged from memory.

Even applications built specifically for iOS 4 should be prepared to handle situations where multitasking is not available. If the presence or absence of multitasking changes the way your application behaves, it can use the `multitaskingSupported` property of the `UIDevice` class to determine whether the feature 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 in 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 declares this support by including the `UIBackgroundModes` key in its `Info.plist` file. 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. (This includes streaming audio or video content using AirPlay.)
- **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 does not include this key, any audio being played by the application stops when the application moves to the background.

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

- **Task completion**—Applications can ask the system for extra time to complete a given task.
- **Local notifications**—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 [“Implementing Long-Running Background Tasks”](#) (page 66).

Implementing Long-Running Background Tasks

Applications can request permission to run in the background in order to manage specific services for the user. Such applications do not run continuously but are woken up by the system frameworks at appropriate times to perform work related to those services.

Tracking the User’s Location

There are several ways to track the user’s location in the background, some of which do not actually involve running regularly in the background:

- **Applications can register for significant location changes. (Recommended)** The significant-change location service offers a low-power way to receive location data and is highly recommended for applications that do not need high-precision location data. With this service, location updates are generated only when the user’s location changes significantly; thus, it is ideal for social applications or applications that provide the user with noncritical, location-relevant information. If the application is suspended when an update occurs, the system wakes it up in the background to handle the update. If the application starts this service and is then terminated, the system relaunches the application automatically when a new location becomes available. This service is available in iOS 4 and later, only on devices that contain a cellular radio.
- **Applications can continue to use the standard location services.** Although not intended for running indefinitely in the background, the standard location services are available in all versions of iOS and provide the usual updates while the application is running, including while running in the background.

However, updates stop as soon as the application is suspended or terminated, and new location updates do not cause the application to be woken up or relaunched. This type of service is appropriate when location data is used primarily when the application is in the foreground.

- **An application can declare itself as needing continuous background location updates.** An application that needs regular location updates, both in the foreground and background, should add the `UIBackgroundModes` key to its `Info.plist` file and set the value of this key to an array containing the `location` string. This option is intended for applications that provide specific services, such as navigation services, that involve keeping the user informed of his or her location at all times. The presence of the key in the application's `Info.plist` file tells the system that it should allow the application to run as needed in the background.

You are encouraged to use the significant location change service or use the standard services sparingly. Location services require the active use of an iOS device's onboard radio hardware. Running this hardware continuously can consume a significant amount of power. If your application does not need to provide precise and continuous location information to the user, it is best to use those services that minimize power consumption. Chief among these low-power services is the significant location change service introduced in iOS 4. This service provides periodic location updates and can even wake up a background application, or relaunch a terminated application, to deliver them.

For applications that require more precise location data at regular intervals, such as navigation applications, you need to declare the application as a continuous background application. This option is available for applications that truly need it, but it is the least desirable option because it increases power usage considerably.

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

Playing Background Audio

Applications that play audio can include the `UIBackgroundModes` key (with the value `audio`) in their `Info.plist` file to register as a background-audio application. This key is intended for use by applications that provide audible content to the user while in the background, such as music-player or streaming-audio applications. Applications that support audio or video playback over AirPlay should also include this key so that they continue streaming their content while in the background.

When the `audio` value is present, the system's media frameworks automatically prevent your application from being suspended when it moves to the background. As long as it is playing audio or video content, the application continues to run in the background to support that content. However, if the application stops playing that audio or video, the system suspends it. Similarly, if the application does not include the appropriate key in its `Info.plist` file, the application becomes eligible for suspension immediately upon entering the background.

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. (For video playback over AirPlay, you must use the Media Player framework to present your video.) Because your application is not suspended while playing media files, callbacks operate normally while your application is in the background. Your application should limit itself to doing only the work necessary to provide data for playback while in the background. For example, a streaming audio application would download any new data from its server and push the current audio samples out for playback. You should not perform any extraneous tasks that are unrelated to playing the content.

Because more than one application may support audio, the system limits which applications can play audio at any given time. The foreground application always has permission to play audio. In addition, one or more background applications may also be allowed to play some audio content depending on the configuration of their audio session object. Applications should always configure their audio session object appropriately and work carefully with the system frameworks to handle interruptions and other types of audio-related notifications. For information on how to configure your application's audio session properly for background execution, see *Audio Session Programming Guide*.

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 so that it can receive incoming calls and other relevant data. Rather than keep VoIP applications awake all the time, the system allows them to be suspended and provides facilities for monitoring their sockets for them. When incoming traffic is detected, the system wakes up the VoIP application and returns control of its sockets to it.

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 one of the application's sockets for VoIP usage.
3. Before moving to the background, call the `setKeepAliveTimeout:handler:` method to install a handler to be executed periodically. Your application can use this handler to maintain its service connection.
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 to deliver audio while in the background. Therefore, you should include both the `audio` and `voip` values to the `UIBackgroundModes` key. If you do not do this, your application cannot play audio while it is in the background. 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 tag your application's main communication socket specifically for VoIP usage. Tagging this socket tells the system that it should take over management of the socket when your application is suspended. The handoff itself is totally transparent to your application. And when new data arrives on the socket, the system wakes up the application and returns control of the socket so that the application can process the incoming data.

You need to tag only the socket you use for communicating with your VoIP service. This is the socket you use to receive incoming calls or other data relevant to maintaining your VoIP service connection. Upon receipt of incoming data, the handler for this socket needs to decide what to do. For an incoming call, you likely want to post a local notification to alert the user to the call. For other noncritical data, though, you might just process the data quietly and allow the system to put your application back into the suspended state.

In iOS, most sockets are managed using streams or other high-level constructs. To configure a socket for VoIP usage, the only thing you have to do beyond the normal configuration is add a special key that tags the interface as being associated with a VoIP service. Table 4-1 lists the stream interfaces and the configuration for each.

Table 4-1 Configuring stream interfaces for VoIP usage

Interface	Configuration
NSInputStream and NSOutputStream	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> .
NSURLRequest	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> .
CFReadStreamRef and CFWriteStreamRef	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.

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 type of exit could happen in cases where there is memory pressure and your application is terminated as a result.) However, terminating the application also releases all of its sockets, including the one used to maintain the VoIP service connection. Therefore, when the application is launched, it always needs to create its sockets 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 in the `applicationDidEnterBackground:` method of your application delegate. Once installed, the system calls your handler at least once before the timeout interval expires, waking up your application as needed to do so.

Your keep-alive handler executes in the background and should 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, specify the largest timeout value that is practical for your application's 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

As with any background audio application, 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 object only when it is needed. For example, you would create it to notify the user of an incoming call or while the user was actually on a call. As soon as the call ends, you would then release the audio session and give other audio applications the opportunity to play their audio.

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

Completing a Finite-Length 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 `backgroundTimeRemaining` property of the `UIApplication` object contains the amount of time the application has to run.)

You can use task completion to ensure that important but potentially long-running operations do not end abruptly when the user leaves the application. For example, you might use this technique to save user data to disk or 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 task is complete and that the application can now be suspended. Because applications are given only a limited amount of time to finish background tasks, you must call this method before time

expires or the system will terminate your application. To avoid termination, you can also provide an expiration handler when starting a task and call the `endBackgroundTask:` method from there. (You can also check the value in the `backgroundTimeRemaining` property of the application object to see how much time is left.)

An application can have any number of 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 must not take too long to execute. By the time your expiration handler is called, your application is already very close to its time limit. For this reason, perform only minimal cleanup of your state information and end the task.

Scheduling the Delivery of Local Notifications

The `UILocalNotification` class in `UIKit` provides a way to schedule the delivery of local notifications. 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 alert 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 configures only one alarm at a time and cancels the previous alarm before scheduling a new one. (Your own applications can have no more than 128 local notifications active at any given time, any of which can be configured to repeat at a specified interval.) The alarm itself consists of an alert box and a sound file 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, μ -Law, or a-Law. You can also specify the sound name `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*.

Implementing Application Preferences

Preferences are application-specific settings used to configure the behavior or appearance of an application. In iOS, the Foundation framework provides the low-level mechanism for storing the actual preference data. Applications then have two options for presenting preferences:

- Display preferences inside the application.
- Use a Settings bundle to manage preferences from the Settings application.

Which option you choose depends on how you expect users to interact with them. The Settings bundle is generally the preferred mechanism for displaying preferences. However, games and other applications that contain configuration options or other frequently accessed preferences might want to present them inside the application instead. Regardless of how you present them, you still use the `NSUserDefaults` class to access preference values from your code.

This chapter focuses on the creation of a Settings bundle for your application. A **Settings bundle** contains files that describe the structure and desired presentation style of your preferences. The Settings application uses this information to create an entry for your application and to display your custom preference pages.

For guidelines on how to manage and present settings and configuration options, see *iOS Human Interface Guidelines*.

The Settings Application 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.

Every application with a Settings bundle 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 create child pages that link off the main page or other child pages. There is no specific limit to the number of child pages you may create, but you should strive to keep your preferences as simple and easy to navigate as possible.

The contents of each page consists of one or more controls that you configure. Table 5-1 lists the types of controls supported by the Settings application and describes how you might use each type. The table also lists the raw key name stored in the configuration files of your Settings bundle.

Table 5-1 Preference control types

Control Type	Description
Text field	The text field type displays a title (optional) 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> .
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> .
Multivalue	The multivalue 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 for organizing 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 76). 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 Settings page files, see [“Creating and Modifying the Settings Bundle”](#) (page 78).

The Settings Bundle

A Settings 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 describe the individual pages of preferences. It may also include other support files needed to display your preferences, such as images or localized strings. Table 5-2 lists the contents of a typical Settings bundle.

Table 5-2 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 name of this file must be <code>Root.plist</code> . The contents of this file are described in more detail in “The Settings Page File Format” (page 76).
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.
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 file. The strings files provide the localized strings to display for 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, the 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 92).

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 build 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 in your preferences page prior to display.

The Settings Page File Format

Each Settings page file 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 78). You can also edit property-list files using the Property List Editor application that comes with the Xcode tools.

Note: Xcode 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.

The root element of each Settings page file contains the keys listed in Table 5-3. Only one key is actually required, but it is recommended that you include both of them.

Table 5-3 Root-level keys of a preferences Settings page file

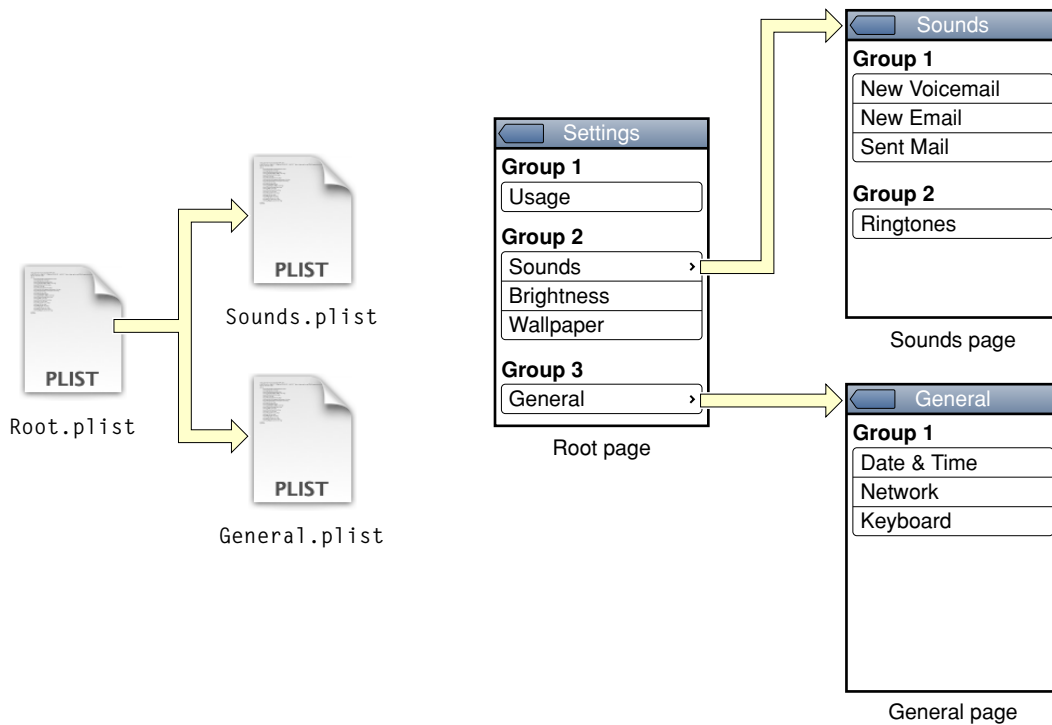
Key	Type	Value
PreferenceSpecifiers (required)	Array	The value for this key is an array of dictionaries, with each dictionary containing the information for a single control. For a list of control types, see Table 5-1 (page 74). For a description of the keys associated with each control, see <i>Settings Application Schema Reference</i> .
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 77).

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 a file called `Root.plist`. 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 control in the parent page. A child pane control creates a row that, when tapped, displays a new page of settings. The `File` key of the child pane control identifies the name of the `.plist` file with 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 control used 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 5-1 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 5-1 Organizing preferences using child panes

For more information about child pane controls 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 iOS applications follow. It first tries to find a localized version of the resource that matches the user's preferred language setting. If no such resource exists, 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*.

Creating 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 expand this bundle as needed 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. Under **iOS**, choose **Resource**, and then select the **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 has the following structure:

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

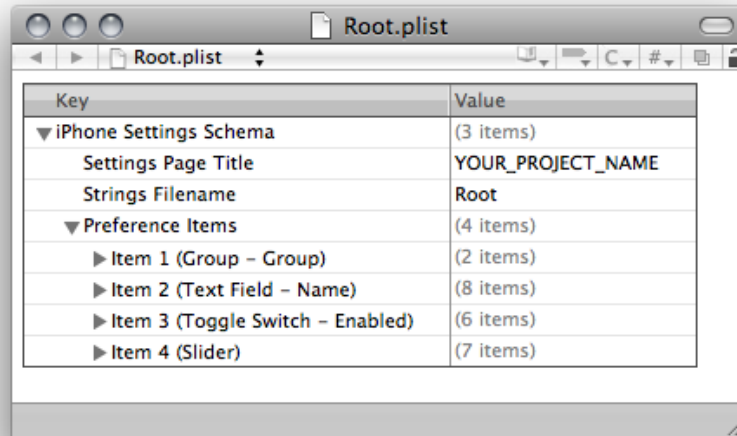
Preparing the Settings Page for Editing

After creating your Settings bundle using the Settings Bundle template, you can format the contents of each Settings page file 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 Settings page 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 5-2) to make it easier for you to understand and edit the file's contents.

Figure 5-2 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 5-3. 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 78) before proceeding with these steps.

Figure 5-3 A root Settings page

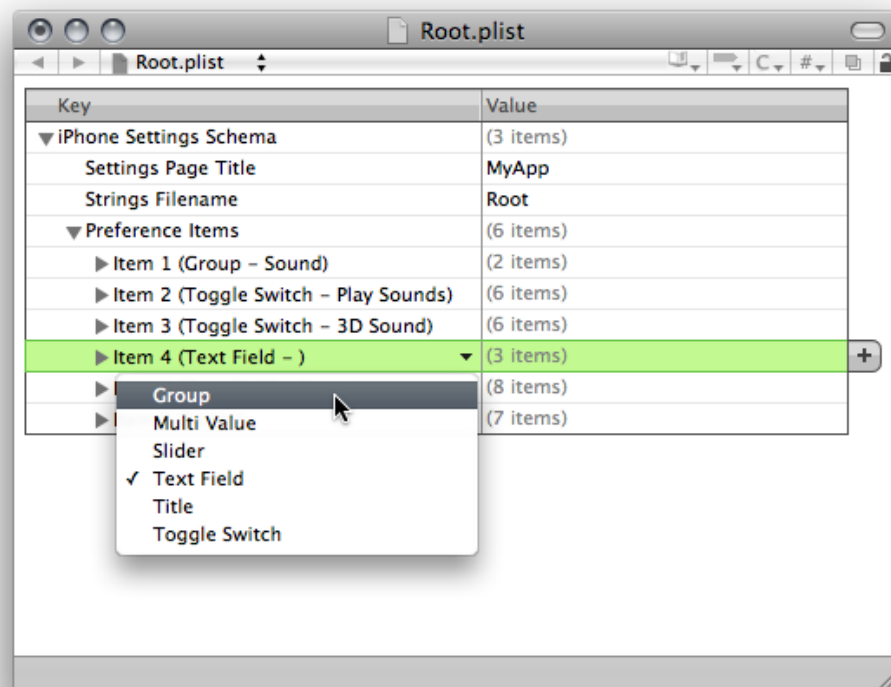
1. Disclose the Preference Items key to display the default items that come with the template.
2. Change the title of Item 0 to Sound.
 - Disclose Item 0 of Preference Items.
 - Change the value of the Title key from Group to Sound.
 - Leave the Type key set to Group.
 - Click the disclosure triangle of the item to hide its contents.
3. Create the first toggle switch for the newly renamed Sound group.
 - Select Item 2 (the toggle switch item) of Preference Items and choose Edit > Cut.
 - Select Item 0 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.
 - Click the disclosure triangle of the item to hide its contents.
4. Create a second toggle switch for the Sound group.
 - Select Item 1 (the Play Sounds toggle switch).

- Choose Edit > Copy.
- Choose Edit > Paste to place 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`.
- Click the disclosure triangle of the item to hide its contents.

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

5. Change Item 3 into a Group control and name it User Info.

- Click Item 3 in the Preferences Items. This displays a pop-up menu with a list of item types.
- From the pop-up menu, choose Group to change the type of the control.



- Disclose the contents of Item 3.
- Set the value of the `Title` key to `User Info`.
- Click the disclosure triangle of the item to hide its contents.

6. Create the Name field.

- Select `Item 4` in the `Preferences Items`.
- Using the pop-up menu, change its type to `Text Field`.
- Set the value of the `Title` key to `User Info`.
- Set the value of the `Identifier` key to `user_name`.
- Click the disclosure triangle of the item to hide its contents.

7. Create the Experience Level settings.

- Select `Item 4` and click the `Add (+)` button (or press `Return`) to create a new item.
- Click the new item and set its type to `Multi Value`.
- Disclose the item's 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 `Add` button to add a `Titles` array.
- Disclose the `Titles` array and click the button at the right edge of the table. Clicking this button adds a new subitem to `Titles`.
- Select the new subitem and click the `Add` button two more times to create a total of three subitems.
- Set the values of the subitems to `Beginner`, `Expert`, and `Master`.
- Hide the key's subitems.
- Click the `Add` button to create the `Values` array.
- Add three subitems to the `Values` array and set their values to `0`, `1`, and `2`.
- Hide the contents of `Item 5`.

8. 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.

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 File.
3. Under Mac OS X, select Resource, and then select the Property List template.
4. Select the new file to display its contents in the editor.
5. Select the Root key in the editor.
6. 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 79). To display the settings for your page, you must reference it from a child pane control as described in ["Hierarchical Preferences"](#) (page 76).

Accessing Your Preferences

An iOS application gets and sets preference 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 5-1 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 79) and assign it to an application-specific instance variable.

Listing 5-1 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*.

Specifying Default Values for Preferences

It is recommended that you register any default preference values programmatically at launch time in addition to including them in your Settings bundle property lists. For newly installed applications, default preference values from the application's Settings bundle are not set until the Settings application runs. This means that if the user runs your application before running Settings, the default values specified in your Settings bundle are unavailable. Setting such values programmatically at launch time ensures that your application always has appropriate values. To register default values programmatically, use the `registerDefaults:` method of the `NSUserDefaults` class.

For information and examples of how to register default preference values programmatically, see “Using `NSUserDefaults`” in *User Defaults Programming Topics*.

Debugging Preferences for Simulated Applications

When running your application, iOS Simulator stores any preferences values for your application in `~/Library/Application Support/iOS 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 build your application, Xcode preserves your application preferences and other relevant library files. If you want to remove the current preferences for testing purposes, you can delete the application from Simulator or choose **Reset Contents and Settings** from the iOS Simulator menu.

Build-Time Configuration Details

Configuring your application properly 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 or the App Store and must be configured properly.

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 file and supporting resource files such as application icons, image files, and localized content. Table 6-1 lists the contents of a typical iOS application bundle, which for demonstration purposes 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 6-1 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 containing configuration data for the application. The system uses this data to determine how to interact with the application at specific times. This file is required. For more information, see “The Information Property List” (page 87).
<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.) This file is optional but recommended. For more information, see “The Information Property List” (page 87)
Application icons	One or more image files containing the icons used to represent your application in different parts of the system. An application icon is required. For information about these image files, see “Application Icons” (page 92).

Files	Description
One or more launch images	<p>A screen-size image that the system displays when it launches your application. 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 94).</p>
iTunesArtwork	<p>The 512 x 512 pixel icon for an application that is distributed using ad hoc distribution. This icon is normally 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), if you were distributing your application that way. The filename must be the one shown at left and must not include a filename extension.)</p> <p>This file is required for ad hoc distribution and optional otherwise.</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.</p> <p>This bundle is optional. For more information about specifying settings, see “Displaying Application Preferences” (page 62).</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 102).</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, include a default version of these files at the top level of your application bundle. The default version is used when 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 your application’s main bundle object (typically using the `NSBundle` class).
2. Use the methods of the bundle object to obtain a file-system path to the desired resource file.
3. Open (or access) the file and use it.

You obtain a reference to your application's main bundle using the `mainBundle` class method of `NSBundle`. The `pathForResource:ofType:` method is one of several `NSBundle` methods that you can use to retrieve the location of resources. The following example shows 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];
```

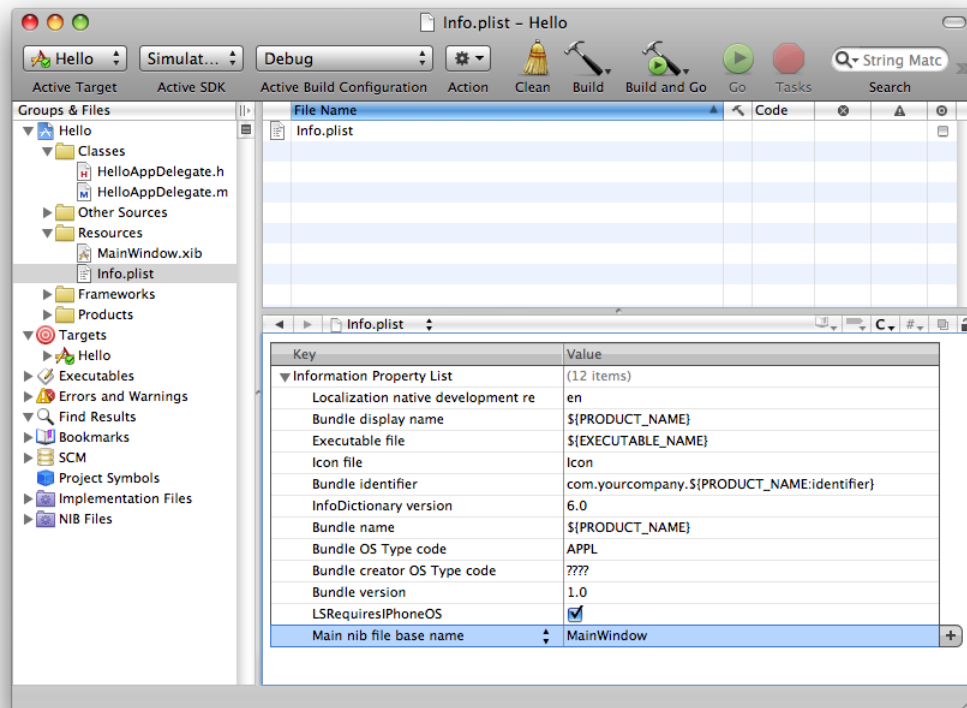
Note: If you prefer to use Core Foundation to access bundles, you can obtain a `CFBundleRef` opaque type using the `CFBundleGetMainBundle` function. You can then use that opaque type plus the Core Foundation functions to locate any bundle resources.

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*.

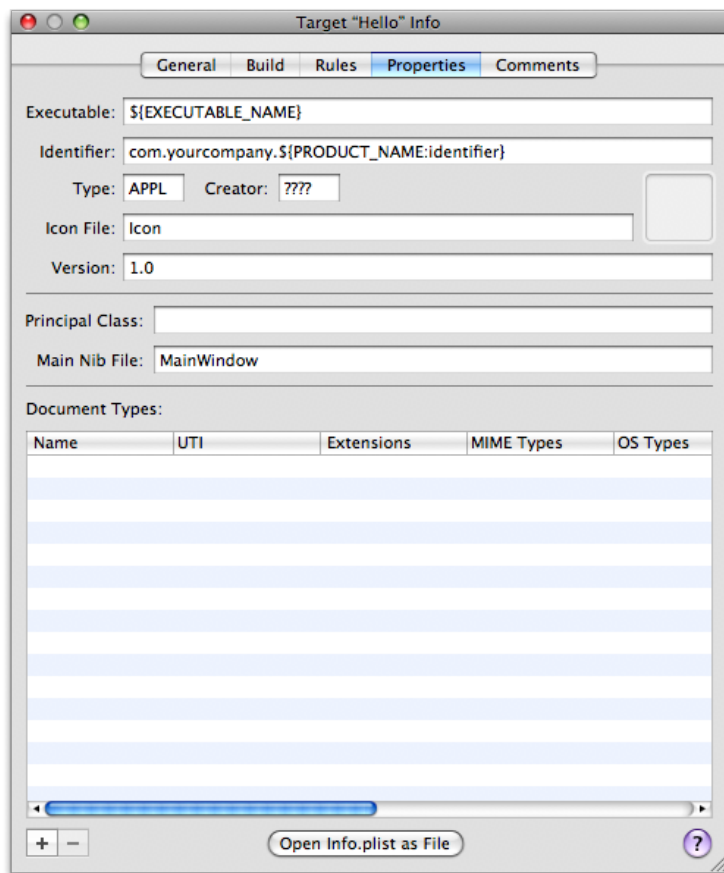
The Information Property List

An information property list (`Info.plist`) file is included with every iOS application project created by Xcode. This file contains essential runtime-configuration information for the application.

To view the contents of your `Info.plist` file, select it in the Groups & files pane. Xcode displays a property list editor similar to the one in Figure 6-1. You can use this window to edit the property values and add new key-value pairs. By default, Xcode displays a more user-friendly version of each key name. To see the actual key names Xcode adds to the `Info.plist` file, Control-click an item in the editor and choose Show Raw Keys/Values from the contextual menu that appears.

Figure 6-1 The information property list editor

In Xcode, you can also edit some application properties using the Info window, as shown in Figure 6-2. This window contains the fundamental information that all applications must provide, such as the executable name and identifier and the name of the main nib file.

Figure 6-2 The Properties pane of a target's Info window

Xcode automatically adds some important keys to the `Info.plist` file of all new projects and sets their initial values. 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 keys 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 webpage, but some of it must be embedded directly in your application bundle.

Declaring the Required Device Capabilities

If your application requires the presence or absence of specific device capabilities in order to run, you must declare those requirements using the `UIRequiredDeviceCapabilities` key in your application's `Info.plist` file. At runtime, iOS cannot launch your application unless the declared 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.

The `UIRequiredDeviceCapabilities` key (supported in iOS 3.0 and later) is normally used to declare the specific capabilities that your application requires. The value of this key is either an array or a dictionary that contains additional keys identifying the corresponding features. If you use an array, the presence of a key indicates that the feature is required; the absence of a key indicates that the feature is not required and that the application can run without it.

If you use a dictionary for the value of the `UIRequiredDeviceCapabilities` key, each key in the dictionary similarly corresponds to one of the targeted features and contains a Boolean value. A value of `true` for a key indicates that the feature is required. However, a value of `false` indicates that the feature must *not* be present on the device. In other words, for features that are optional, you should omit the key entirely rather than include it and set its value to `false`.

Table 6-2 lists the keys that you can include in the array or dictionary for the `UIRequiredDeviceCapabilities` key. You should include keys only for the features that your application absolutely requires. If your application can accommodate the lack of a specific feature, do not include the corresponding key.

Table 6-2 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.

Key	Description
auto-focus-camera	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.
front-facing-camera	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.
camera-flash	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.
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 location data more accurate than the cell or Wi-fi radios might otherwise provide.
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.

Key	Description
<code>opengles-2</code>	Include this key if your application requires (or specifically prohibits) the presence of the OpenGL ES 2.0 interfaces.
<code>armv6</code>	Include this key if your application is compiled only for the armv6 instruction set. (iOS 3.1 and later)
<code>armv7</code>	Include this key if your application is compiled only for the armv7 instruction set. (iOS 3.1 and later)
<code>peer-peer</code>	Include this key if your application requires (or specifically prohibits) peer-to-peer connectivity over a Bluetooth network. (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 provide an icon to be displayed on a device's Home screen and in the App Store. Application may actually specify several different icons for use in different situations. For example, applications can provide a small icon to use when displaying search results and provide a high-resolution icon for devices with Retina displays.

To specify the icons for your application, add the `CFBundleIconFiles` key to your application's `Info.plist` file and add the filenames of your icon image to 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 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 6-3 lists the dimensions of the icons you can associate with the `CFBundleIconFiles` key, along with the intended usage for each one. For applications that run on devices with Retina displays, two versions of each icon 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. You can find out more about specifying and loading high-resolution image resources in *Drawing and Printing Guide for iOS*. For detailed information about the usage and preparation of your icons, see *iOS Human Interface Guidelines*.

Table 6-3 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 devices with Retina displays only.
Application icon (required)	iPad	72 x 72 pixels	This is the main icon for applications running on iPad.

Icon	Idiom	Size	Usage
Settings/Search results icon	iPhone/iPad	29 x 29 pixels 58 x 58 pixels (@2x)	This is the icon displayed in conjunction with search results on iPhone and iPod touch. This icon is also used by the Settings application on all devices. (The @2x variant of the icon is for use on devices with Retina displays only and is not supported on iPad.)
Search results icon	iPad	50 x 50 pixels	This is the icon 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 the names of all 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, even if they are not included in the array.

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 supported only in iOS 3.2 and later.

If your iPhone application is running in iOS 3.1.3 or earlier, the system does not look for the `CFBundleIconFiles` key and instead looks for icon files with specific names. The `CFBundleIconFiles` key was introduced in iOS 3.2 and is not recognized by earlier versions of the system. Although the sizes of the icons are the same as those in [Table 6-3](#) (page 92), if your application supports deployment on iOS 3.1.3 and earlier, you must use the following filenames when naming your icons:

- `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 submit to the App Store (typically a JPEG or PNG file), if you were distributing 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 *iOS Human Interface Guidelines*.

Application Launch Images

When the system launches an application, it temporarily displays a static launch image on the screen. Your application provides this image, with the image contents usually containing a prerendered version of your application's default user interface. The purpose of this image is to give the user immediate feedback that the application launched, but it 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 system removes the image and displays your application's windows and views.

Every application must provide at least one launch image. This image is typically in 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. All 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 a custom string that you specify using the `UILaunchImageFile` key in your application's `Info.plist` file. The `<scale_modifier>` portion is the optional string `@2x` and should be included only for images intended for use on Retina displays. Other optional modifiers may also be included in the name, and several standard modifiers are discussed in the sections that follow.

Table 6-4 lists the 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 *iOS Human Interface Guidelines*.

Table 6-4 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 (320 x 480). 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.

For more information about the `UILaunchImageFile` key, see *Information Property List Key Reference*.

Providing Launch Images for Different Orientations

In iOS 3.2 and later, an iPad application can provide both landscape and portrait versions of its launch images. Each orientation-specific launch image must include a special modifier string in its filename. The format for orientation-specific launch image filenames is as follows:

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

Table 6-5 lists the possible modifiers you can specify for the `<orientation_modifier>` value in your image filenames. As with all launch images, each 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 6-5 Launch image orientation modifiers

Modifier	Description
-PortraitUpsideDown	Specifies an upside-down portrait version of the launch image. A file with this modifier takes precedence over a file with the -Portrait modifier for this specific orientation.
-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. Before iOS 3.2, 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 iPad application would be named `MyLaunchImage-Landscape.png` and `MyLaunchImage-Portrait.png`. If you do 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 displayed 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 the `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 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

Universal applications must provide launch images for both the iPhone and iPad idioms. Because iPhone applications require only one launch image (`Default.png`), whereas iPad applications typically require different images for portrait and landscape orientations, you can usually do without device-specific modifiers. However, if you create multiple launch images for each idiom, the names of device-specific image files are likely to collide. In that situation, you can append a device modifier to filenames to indicate they are for a specific platform only. The following device modifiers are recognized for launch images in iOS 4.0 and later:

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

Because device modifiers are not supported in iOS 3.2, the minimal set of launch images needed for a universal application (running in iOS 3.2 and later) would need to be named `Default.png` and `Default~iphone.png`. In that case, the `Default.png` file would contain the iPad launch image (for all orientations) and the `Default~iphone.png` file would contain the iPhone version of the image. (To support high-resolution displays, you would also need to include a `Default@2x~iphone.png` launch image.)

Note: If you are using the `UILaunchImageFile` key in your `Info.plist` file to specify a custom base name for your launch image files, you can include a device-specific version of that key to specify a different base name for each device type. For example, you could specify a `UILaunchImageFile~ipad` key to specify a different base name for iPad launch images. Specifying different base names lets an application that supports multiple devices avoid naming conflicts among its launch images. 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 one or more custom URL schemes, it can also provide a custom launch images for each URL scheme. When the system launches your application to handle a URL, it displays the launch image associated with the scheme of the given URL. In this case, the format for your launch image filenames is as follows:

`<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`, the system looks for an image with the name `Default-myscheme.png` (or `Default-myscheme@2x.png` for Retina displays) in the application's bundle. If the application's `Info.plist` file includes the `UILaunchImageFile` key, 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 filename is `<basename>-<url_scheme><orientation_modifier><scale_modifier><device_modifier>.png`. For more information about the launch orientation modifiers, see [“Providing Launch Images for Different Orientations”](#) (page 95).

In addition to including the launch images at the top level of your bundle, you can also include localized versions of your launch images in your application’s language-specific project subdirectories. For more information on localizing resources in your application, see [“Internationalizing Your Application”](#) (page 102).

Creating a Universal Application

A universal application is a single application that is optimized for iPhone, iPod touch, and iPad devices. Providing a single binary that adapts to the current device offers the best user experience but, of course, involves extra work on your part. Because of the differences in device screen sizes, most of your window, view, and view controller code for iPad is likely to differ from the code for iPhone and iPod touch. In addition, there are things you must do to ensure your application runs correctly on each device type.

The following sections highlight the key changes you must make to an existing application to ensure that it runs smoothly on any type of device.

Configuring Your Xcode Project

The first step to creating a universal application is to configure your Xcode project. If you are creating a new project, you can create a universal application using the Window-based application template. If you are updating an existing project, you can use Xcode’s Upgrade Current Target for iPad command to update your project.

1. Open your Xcode project.
2. In the Targets section, select the target you want to update to a universal application.
3. Choose Project > Upgrade Current Target for iPad and follow the prompts to create one universal application.

Xcode updates your project by modifying several build settings to support both iPhone and iPad.

Important: You should always use the Upgrade Current Target for iPad command to migrate existing projects. Do not try to migrate files manually.

The main change Xcode makes to your project is to set the Targeted Device Family build setting to iPhone/iPad. The Base SDK of your project is also typically changed to an appropriate value that supports both device types. The deployment target of your project should remain unchanged.

In addition to updating your build settings, Xcode creates a new version of the application's main nib file and updates the application's `Info.plist` file to support the iPad version of the main nib file. Only the main nib file is created. Xcode does not create new versions of any other nib files in your project. Because these files tend to be tied to your application's view controllers, it is up to you to define both the views and view controllers you want to use on iPad.

When running in versions of iOS that support only iPad or only iPhone and iPod touch, your application must be careful what symbols it uses. For example, an application trying to use the `UISplitViewController` class while running in iOS 3.1 would crash because the symbol would not be available. To avoid this problem, your code must perform runtime checks to see whether a particular symbol is available for use. For information about how to perform the needed runtime checks, see [“Adding Runtime Checks for Newer Symbols”](#) (page 99).

Updating Your Info.plist Settings

Most of the existing keys in a universal application's `Info.plist` file should remain the same to ensure that your application behaves properly on iPhone and iPod touch devices. However, you should add the `UISupportedInterfaceOrientations` key to your `Info.plist` to support iPad devices. Depending on the features of your application, you might also want to add other keys introduced in iOS 3.2.

If you need to configure your application differently for the iPad and iPhone idioms, you can specify device-specific values for `Info.plist` keys in iOS 3.2 and later. When reading the keys of your `Info.plist` file, the system interprets each key using the following format:

key_root - *<platform>* ~ *<device>*

In this format, the *key_root* portion represents the original name of the key. The *<platform>* and *<device>* portions are both optional endings that you can use for keys that are specific to a platform or device. Specifying the string `iphoneos` for the platform indicates that the key applies to all iOS applications. (Of course, if you are deploying your application only to iOS anyway, you can omit the platform portion altogether.) To apply a key to a specific device, use one of the following values:

- `iphone`. The key applies to iPhone devices.
- `ipod`. The key applies to iPod touch devices.
- `ipad`. The key applies to iPad devices.

For example, to indicate that you want your application to launch in a portrait orientation on iPhone and iPod touch devices but in landscape-right on iPad, you would configure your `Info.plist` with the following keys:

```
<key>UIInterfaceOrientation</key>
<string>UIInterfaceOrientationPortrait</string>
<key>UIInterfaceOrientation~ipad</key>
<string>UIInterfaceOrientationLandscapeRight</string>
```

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

Updating Your View Controllers and Views

Of all the changes you must make to create a universal application, updating your view controllers and views is the biggest. Because of the different screen sizes, you may need to completely redesign your existing interface to support both types of device. For example, you need to create separate sets of view controllers (or modify your existing view controllers) to support the different view sizes.

For views, the main modification is to redesign your view layouts to support the larger screen. Simply scaling existing views may work but often does not yield the best results. Your new interface should make use of the available space and take advantage of new interface elements where appropriate. Doing so is more likely to result in an interface that feels more natural to the user and not just an iPhone application on a larger screen.

Here are some additional things you must consider when updating your view and view controller classes:

- **For view controllers:**
 - If your view controller uses nib files, you must specify different nib files for each device type when creating the view controller.
 - If you create your views programmatically, you must modify your view-creation code to support both device types.
- **For views:**
 - If you implement the `drawRect:` method for a view, your drawing code needs to be able to draw to different view sizes.
 - If you implement the `layoutSubviews` method for a view, your layout code must be adaptable to different view sizes.

For information about the view controllers you can use in your applications, see *View Controller Programming Guide for iOS*.

Adding Runtime Checks for Newer Symbols

Any application that supports a range of iOS versions must use runtime checks to protect code that uses symbols introduced in newer versions of the operating system. Thus, if you use the iOS 4.2 SDK to develop applications that run in iOS 3.1 and later, runtime checks allow you to use newer features when they are available and follow alternate code paths when they are not. Failure to include such checks results in crashes when your application tries to use symbols that are not available.

There are several types of checks that you can make:

- Applications that link against iOS SDK 4.2 and later can use the weak linking support introduced in that version of the SDK. This support lets you check for the existence of a given Class object to determine whether you can use that class. For example:

```
if ([UIPrintInteractionController class]) {  
    // Create an instance of the class and use it.  
}
```

```
else {
    // The print interaction controller is not available.
}
```

To use this feature, you must build your application using LLVM and Clang and the application's deployment target must be set to iOS 3.1 or later.

- Applications that link against iOS SDK 4.1 and earlier must use the `NSClassFromString` function to see whether a class is defined. If the function returns a value other than `nil`, you may use the class. For example:

```
Class splitVCClass = NSClassFromString(@"UISplitViewController");
if (splitVCClass)
{
    UISplitViewController* mySplitViewController = [[splitVCClass alloc] init];
    // Configure the split view controller.
}
```

- To determine whether a method is available on an existing class, use the `instancesRespondToSelector:` class method.
- To determine whether a C-based function is available, perform a Boolean comparison of the function name to `NULL`. If the result is `YES`, you can use the function. For example:

```
if (UIGraphicsBeginPDFPage != NULL)
{
    UIGraphicsBeginPDFPage();
}
```

For more information and examples of how to write code that supports multiple deployment targets, see *SDK Compatibility Guide*.

Using Runtime Checks to Create Conditional Code Paths

If your code needs to follow a different path depending on the underlying device type, you can use the `userInterfaceIdiom` property of `UIDevice` to determine which path to take. This property provides an indication of the style of interface to create: iPad or iPhone. Because this property is available only in iOS 3.2 and later, you may need to check for the availability of this property before accessing it. The simplest way to do this is to use the `UI_USER_INTERFACE_IDIOM` macro.

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    // The device is an iPad running iOS 3.2 or later.
}
else {
    // The device is an iPhone or iPod touch.
}
```

Updating Your Resource Files

Because resource files are generally used to implement your application's user interface, you need to make the following changes:

- In addition to the `Default.png` file displayed when your application launches on iPhone devices, you must add new launch images for iPad devices as described in [“Providing Launch Images for Different Orientations”](#) (page 95).
- If you use images, you may need to add larger (or higher-resolution) versions to support iPad devices.
- If you use nib files, you need to provide a new set of nib files for iPad devices.
- Your application icons must be sized appropriately for iPad, as described in [“Application Icons”](#) (page 92).

When using different resource files for each platform, you can conditionally load those resources just as you would conditionally execute code. For more information about how to use runtime checks, see [“Using Runtime Checks to Create Conditional Code Paths”](#) (page 100).

Using a Single Xcode Project to Build Two Applications

Using a single Xcode project for both the iPhone and iPad versions of your application simplifies the development process tremendously by allowing the applications to share common code. The Project menu in Xcode includes an Upgrade Current Target for iPad command that makes it easy to add a target for iPad devices to your existing iPhone project. To use this command, do the following:

1. Open the Xcode project for your existing iPhone application.
2. Select the target for your iPhone application.
3. Choose Project > Upgrade Current Target for iPad and follow the prompts to create two device-specific applications.

Important: You should always use the Upgrade Current Target for iPad command to migrate existing projects. Do not try to migrate files manually.

The Upgrade Current Target for iPad command creates a new iPad target and creates new nib files for your iPad project. The nib files are based on the existing nib files already in your project but the windows and top-level views in those nib files are sized for the iPad screen. Although the top-level views are resized, the command does not attempt to modify the size or position of any embedded subviews, instead leaving your view layout essentially the same as it was. It is up to you to adjust the layout of those embedded views.

Creating a new target is also just the first step in updating your project. In addition to adjusting the layout of your new nib files, you must update your view controller code to manage those nib files. In nearly all cases, you will want to define a new view controller class to manage the iPad version of your application interface, especially if that interface is at all different from your iPhone interface. You can use conditional compilation to coordinate the creation of the different view controllers. If you make few or no changes to your view hierarchy, you could also reuse your existing view controller class. In such a situation, you would similarly use conditional compilation to initialize your view controller with the appropriate nib file for the underlying device type.

In addition to your view controllers, any classes that are shared between iPhone and iPad devices need to include conditional compilation macros to isolate device-specific code. Although you could also use runtime checks to determine whether specific classes or methods are available, doing so would only increase the size of your executable by adding code paths that would not be followed on one device or the other. Letting the compiler remove this code helps keep your code cleaner.

Beyond conditionally compiling your code for each device type, you should feel free to incorporate whatever device-specific features you feel are appropriate. Features that are supported only on iPad devices are described elsewhere in this document. Any code you write using these features must be run only on iPad devices.

For more information on using conditional compilation and the availability macros, see *SDK Compatibility Guide*.

Internationalizing Your Application

The process of preparing a project to handle content in different languages is called internationalization. The process of converting text, images, and other content into other languages is called *localization*. Project resources 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 their preferred language from the Language preferences view (see Figure 6-3). They get to this view from the General > International settings.

Figure 6-3 The Language preference view

Your application bundle can include multiple language-specific resource directories. The names of these directories consist of 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 directories 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*.

Each `lproj` directory contains a localized copy of the application’s resource files. When you request the path to a resource file using the `NSBundle` class or the `CFBundleRef` opaque type, the path you get back reflects the resource for the user’s preferred language.

For more information about internationalization and how you support localized content 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 than that for Mac OS X applications. The following sections describe the factors you should consider throughout the development process.

Do Not Block the Main Thread

Be sure to limit the type of work you do on 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 or potentially unbounded tasks, 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 use Grand Central Dispatch (GCD) or operation objects to perform tasks asynchronously.

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 similarly kill the application before it has a chance to write out crucial user data.

For more information about using GCD, 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 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. In addition, applications running under multitasking must share system memory with all other running applications. 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 suspended applications, code pages, or other nonvolatile resources from memory. However, removing those applications and resources from memory may be only a temporary fix, especially if they 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.

Observing Low-Memory Warnings

When the system dispatches a low-memory warning to your application, respond immediately. iOS notifies all running applications whenever the amount of free memory dips below a safe threshold. (It does not notify suspended applications.) If your application receives this warning, it must free up as much memory as possible. The best types of memory to release are caches and image objects that can be recreated later if needed.

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 default behavior of the `UIViewController` class is to purge 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 data model includes known purgeable resources, you can have a corresponding manager object register for the `UIApplicationDidReceiveMemoryWarningNotification` notification and release its purgeable resources directly. Handling this notification directly avoids the need to route all memory warning calls through the application delegate.

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

Reduce Your Application's Memory Footprint

Table 7-1 lists some tips on how to reduce your application's overall memory footprint. Starting off with a low footprint gives you more room for expanding your application later.

Table 7-1 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 have no memory leaks. You can use the Instruments application to track down leaks in your code, both in 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 disk but must be loaded into memory before they can be used. Property list files and images can be made smaller 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.)

Tip	Actions to take
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 framework 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 wastes memory for no good purpose.
Build your program using the Thumb option.	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 ARMv6, you should disable the Thumb option. If you are building your code for ARMv7, you should leave Thumb enabled.

Allocate Memory Wisely

iOS applications use a managed memory model, in which you must explicitly retain and release objects. Table 7-2 lists tips for allocating memory inside your program.

Table 7-2 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 current 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 a large resource file 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: If you build your application for ARMv6 and your code uses floating-point math extensively, 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 ARMv7, 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. 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 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 being turned 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 generally takes less power to perform a set of calculations all at once than it does to perform them in small chunks over an extended period of time. Doing small bits of work periodically requires waking up the CPU more often and getting it into a state where it can perform your tasks.
- 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-intensive operation you can perform. You can minimize that time 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 simply is 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 cellular 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 iOS Simulator. For example, you can use 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 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 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 drive, 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, and 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 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, especially if the amount of data you are manipulating could grow to more than a few megabytes.
- 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 for communicating over the radio hardware of iOS 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

Implementing code to receive or transmit data across the network is one of the most power-intensive operations on a device. Minimizing the amount of time spent transmitting or receiving data 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 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. Depending on the radio, though, doing so can take several seconds. 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, packets can be lost at any time. Therefore, 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 immediately invokes browsing callbacks when it receives a notification that a service is going away, but network services can disappear without notification. This situation 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 dialog 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 running 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 or is suspended, 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 87).

The Airplane Mode Alert

If your application launches while the device is in airplane mode, the system may display an alert to notify the user of that fact. The system displays this alert 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
2011-02-24	Added information about using AirPlay in the background.
2010-12-13	Made minor editorial changes.
2010-11-15	Incorporated additional iPad-related design guidelines into this document.
	Updated the information about how keychain data is preserved and restored.
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.

Date	Notes
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.
	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>iPhone OS Programming Guide</i> .
2008-07-08	New document that describes iOS and the development process for iPhone applications.