iPhone OS Technology Overview

General
# Contents

## Introduction

**Introduction** 7

- Who Should Read This Document 7
- Organization of This Document 8
- Getting the iPhone SDK 8
- Providing Feedback 8
- See Also 9

## Chapter 1

**About iPhone OS Development** 11

- The iPhone OS Architecture 11
- What’s in the iPhone SDK? 12
- What Can You Create? 13
- How to Use the Reference Library 14

## Chapter 2

**iPhone OS Technologies** 17

- Cocoa Touch Layer 17
  - Apple Push Notification Service 17
  - Address Book UI Framework 18
  - In App Email 18
  - Map Kit Framework 18
  - Peer to Peer Support 19
  - UIKit Framework 19
- Media Layer 20
  - Graphics Technologies 20
  - Audio Technologies 21
  - Video Technologies 22
- Core Services Layer 23
  - Address Book 23
  - Core Data 23
  - Core Foundation 23
  - Core Location 24
  - Foundation Framework 24
  - In App Purchase 25
  - SQLite 25
  - XML Support 25
- Core OS Layer 26
  - CFNetwork 26
  - Accessory Support 26
  - Security 27
  - System 27
Chapter 3  Migrating from Cocoa  29

General Migration Notes  29
 Migrating Your Data Model  29
 Migrating Your User Interface  30
 Memory Management  30
 Framework Differences  30
 UIKit Versus AppKit  31
 Foundation Framework Differences  33
 Changes to Other Frameworks  34

Appendix A  iPhone OS Developer Tools  37

 Xcode  37
 Interface Builder  39
 Instruments  40
 Shark  41

Appendix B  iPhone OS Frameworks  43

 Device Frameworks  43
 Simulator Frameworks  45
 System Libraries  45

Document Revision History  47
Figures and Tables

Chapter 1  
**About iPhone OS Development**  11

| Figure 1-1 | Applications layered on top of iPhone OS  | 12 |
| Figure 1-2 | The iPhone Reference Library              | 15 |
| Figure 1-3 | The Xcode Research Assistant             | 16 |

Chapter 2  
**iPhone OS Technologies**  17

| Figure 2-1 | Layers of iPhone OS                      | 17 |
| Table 2-1   | Core Audio frameworks                    | 22 |

Chapter 3  
**Migrating from Cocoa**  29

| Table 3-1  | Differences in interface technologies   | 31 |
| Table 3-2  | Foundation technologies unavailable in iPhone OS | 33 |
| Table 3-3  | Differences in frameworks common to iPhone OS and Mac OS X | 34 |

Appendix A  
**iPhone OS Developer Tools**  37

| Figure A-1 | An Xcode project window                  | 38 |
| Figure A-2 | Running a project from Xcode             | 39 |
| Figure A-3 | Building iPhone interfaces using Interface Builder | 40 |
| Figure A-4 | Using Instruments to tune your application | 41 |

Appendix B  
**iPhone OS Frameworks**  43

| Table B-1 | Device frameworks                       | 43 |
Introduction

iPhone OS is the operating system at the heart of iPhone and iPod touch devices.

The iPhone OS platform was built using the knowledge that went into the creation of Mac OS X, and many of the tools and technologies used for development on the platform have their roots in Mac OS X as well. Despite its similarities to Mac OS X, you do not need to be an experienced Mac OS X developer to write applications for iPhone OS. The iPhone Software Development Kit (SDK) provides everything you need to get started creating iPhone applications.

Who Should Read This Document

*iPhone OS Technology Overview* is an introductory guide for anyone who is new to the iPhone OS platform. It provides an overview of the technologies and tools that have an impact on the development process and provides links to relevant documents and other sources of information. You should use this document to do the following:

- Orient yourself to the iPhone OS platform.
- Learn about iPhone OS software technologies, why you might want to use them, and when.
- Learn about development opportunities for the platform.
- Get tips and guidelines on how to move to iPhone OS from other platforms.
Find key documents relating to the technologies you are interested in.

This document does not provide information about user-level system features or about features that have no impact on the software development process.

New developers should find this document useful for getting familiar with iPhone OS. Experienced developers can use it as a road map for exploring specific technologies and development techniques.

Organization of This Document

This document has the following chapters and appendixes:

- “About iPhone OS Development” (page 11) provides a high level overview of iPhone OS and developing applications for it using the iPhone SDK.
- “iPhone OS Technologies” (page 17) provides a look at the technology layers in iPhone OS and the features they provide to your applications.
- “Migrating from Cocoa” (page 29) provides starter advice for developers who are migrating an existing Cocoa application to iPhone OS.
- “iPhone OS Frameworks” (page 43) describes the frameworks you can use to develop your software. Use this information to find specific technologies or to find when a given framework was introduced to iPhone OS.
- “iPhone OS Developer Tools” (page 37) provides an overview of the available applications you can use to create software for iPhone OS.

Getting the iPhone SDK

The iPhone SDK contains the tools needed to design, create, debug, and optimize software for iPhone OS. It also contains header files, sample code, and documentation for the platform's technologies. You can download the iPhone SDK from the members area of the iPhone Dev Center, which is located at http://developer.apple.com/iphone. Registration is required but free.

For additional information about the tools available for working with Mac OS X and its technologies, see “iPhone OS Developer Tools” (page 37).

Providing Feedback

If you have feedback about the documentation, you can provide it using the built-in feedback form at the bottom of every page.

If you encounter bugs in Apple software or documentation, you are encouraged to report them to Apple. You can also file enhancement requests to indicate features you would like to see in future revisions of a product or document. To file bugs or enhancement requests, go to the Bug Reporting page of the ADC website, which is at the following URL:
http://developer.apple.com/bugreporter/

You must have a valid ADC login name and password to file bugs. You can obtain a login name for free by following the instructions found on the Bug Reporting page.

See Also

The following documents provide key information related to iPhone development:

- **Cocoa Fundamentals Guide** provides fundamental information about the design patterns and practices used to develop iPhone applications.
- **iPhone Application Programming Guide** provides an architectural overview of iPhone applications along with practical guidance on how to create them.
- **iPhone Human Interface Guidelines** provides guidance and important information about how to design your iPhone application's user interface.
- **iPhone Development Guide** provides important information about the iPhone development process from the tools perspective. This document covers the configuration of devices and the use of Xcode (and other tools) for building, running, and testing your software.
- **The Objective-C Programming Language** introduces Objective-C and the Objective-C runtime system, which is the basis of much of the dynamic behavior and extensibility of iPhone OS.
iPhone OS is the operating system that runs on iPhone and iPod touch devices. This operating system manages the device hardware and also provides the basic technologies needed to implement native applications on the phone. Depending on whether it is installed on an iPhone or an iPod touch, the operating system also ships with several system applications, such as Phone, Mail, and Safari, that provide standard system services for the user.

The iPhone SDK contains the tools and interfaces needed to develop, install, and run custom native applications. Native applications are built using the iPhone OS system frameworks and the Objective-C language and they run directly on iPhone OS. Unlike web applications, native applications are installed physically on a device and can run with or without the presence of a network connection. They reside next to other system applications and both the application and any user data is synced to the user’s computer through iTunes.

The iPhone OS Architecture

The iPhone OS architecture is similar to the basic architecture found in Mac OS X. At the high level, iPhone OS acts as an intermediary between the iPhone and iPod touch hardware and the applications that appear on the screen, as shown in Figure 1-1. Applications that you create never interact directly with the hardware but instead go through system interfaces, which interact with the appropriate drivers. This abstraction protects your application from changes to the underlying hardware.
Figure 1-1 Applications layered on top of iPhone OS

Note: Even though your application is generally protected from changes to the underlying hardware, you still need to account for differences between iPhone and iPod touch devices in your code.

The iPhone OS uses a fairly straightforward software stack. At the very bottom of this stack is the Mach kernel and hardware drivers, which manage the overall execution of programs on the device. On top of that layer are additional layers that contain the core technologies and interfaces you use for development. Although iPhone OS does not expose any of the kernel or driver-level interfaces, it does expose technologies at the higher levels of the stack. For more information about the exposed technologies, see “iPhone OS Technologies” (page 17).

What’s in the iPhone SDK?

The iPhone SDK comes with all of the interfaces, tools, and resources needed to develop iPhone applications from your Intel-based Macintosh computer.

Apple delivers most of its system interfaces in special packages called frameworks. A framework is a directory that contains a dynamic shared library and the resources (such as header files, images, helper applications, and so on) needed to support that library. To use frameworks, you link them into your application project just as you would any other shared library. Linking them to your project gives you access to the features of the framework and also lets the development tools know where to find the header files and other framework resources.
In addition to frameworks, Apple also delivers some technologies in the form of standard shared libraries. Because iPhone OS is based on UNIX, many of the technologies that form the lower-levels of the operating system are derived from open-source technologies. The interfaces for these technologies are therefore available in the standard library and interface directories.

Some of the other key components of the SDK include the following:

- **Xcode Tools** - provides the tools that support iPhone application development, including the following key applications:
  - **Xcode** - an integrated development environment that manages your application projects and lets you edit, compile, run, and debug your code. Xcode integrates with many other tools and is the main application you use during development.
  - **Interface Builder** - a tool you use to assemble your user interface visually. The interface objects you create are then saved to a special resource file format and loaded into your application at runtime.
  - **Instruments** - a runtime performance analysis and debugging tool. You can use Instruments to gather information about your application's runtime behavior and identify potential problems.

- **iPhone Simulator** - a Mac OS X application that simulates the iPhone technology stack, allowing you to test iPhone applications locally on your Intel–based Macintosh computer.

- **iPhone Reference Library** - the SDK includes the reference documentation for iPhone OS by default and you can download a more complete version of the iPhone Reference Library (including sample code and conceptual documentation) by subscribing to the iPhone OS Library doc set. From Xcode, choose Help > Documentation and then click the subscribe button next to the iPhone OS Library doc set in the left-hand column.

Although the SDK provides the software you need to write applications, Xcode and Instruments also let you interact directly with an attached device to run and debug your code on the target hardware. Development on an actual device requires signing up for Apple's paid iPhone Developer Program and configuring a device for development purposes. For more information about the iPhone Developer Program, go to [http://developer.apple.com/iphone/program/](http://developer.apple.com/iphone/program/).

For information on how to install the iPhone SDK and use it for developing iPhone applications, see *iPhone Development Guide*. For more information about the frameworks in iPhone OS, and for information about where to find the low-level system libraries, see “iPhone OS Frameworks” (page 43).

### What Can You Create?

Users can run two different types of custom applications on a device: web applications and native applications. Web applications use a combination of HTML, cascading style sheets, and JavaScript code to implement interactive applications that live on a web server and are transmitted over the network and run inside the Safari web browser. Native applications, on the other hand, are installed directly on the device and can run without the presence of a network connection.

The iPhone SDK supports the creation of native foreground applications that appear on the device's Home screen only. It does not support the creation of other types of code, such as drivers, background applications, frameworks, or dynamic libraries. If you want to integrate code from a framework or dynamic library into your application, you should link that code statically into your application's executable file when building your project.
How to Use the Reference Library

The iPhone Reference Library contains documentation, sample code, tutorials, and more to help you start writing iPhone applications. Because the reference library contains thousands of pages of documentation, ranging from high-level getting started documents to low-level API reference documents, understanding how to find the information is an important step in the development process. The reference library uses a few different techniques for organizing content that should make it easier to browse.

You can access the iPhone Reference Library from the Apple Developer Connection website or from Xcode. In Xcode, choosing Help > Documentation displays the Xcode documentation window, which is the central resource for accessing information about iPhone development. You can use this window to browse the documentation, perform searches, and bookmark documents you may want to refer to later. Documents are grouped by content into doc sets to facilitate updates and to scope searches to only the relevant set of documents.

Before you start writing iPhone applications, you should subscribe to the iPhone OS Library doc set and spend a little time browsing through its contents. The iPhone Reference Library contains a lot of information so it is worth becoming at least somewhat familiar with its layout. Figure 1-2 shows the main page of the reference library in the Xcode documentation window. This page provides quick links for accessing featured content and introductory material. The left-hand column provides more targeted links at the library’s contents. You can browse the library by topic, by framework, or by the type of resource you are looking for. You can also use the Search field in the top right corner of the window to search the contents of the reference library for specific information.
Important: The content of the iPhone Reference Library is updated regularly, so subscribing ensures that you always have the latest information. Subscribing also downloads the contents of the library to your desktop, but you can also access the latest documentation, release notes, Tech Notes, Technical Q&As, and sample code from the iPhone Dev Center (http://developer.apple.com/iphone). All documents are available in HTML and most are also available in PDF format.

Because the reference library provides a tremendous amount of information, sorting through all that information while you are trying to write code can be cumbersome. To help you find specific information quickly, Xcode also provides the Research Assistant, shown in Figure 1-3. You display this window in Xcode by choosing Help > Show Research Assistant and leave it open on your desktop while you write your code. As you type, Xcode constantly updates the contents of this window based on the current selection or text insertion point. It shows you information about the designated symbol, including its syntax and description, and also shows you any related documentation and sample code resources. Clicking the links in this window takes you to the corresponding resource in the reference library.
Figure 1-3 The Xcode Research Assistant

For more information about using the Documentation window and Research Assistant, see Xcode Workspace Guide.
The implementation of iPhone OS technologies can be viewed as a set of layers, which are shown in Figure 2-1. At the lower layers of the system are the fundamental services on which all applications rely, while higher-level layers contain more sophisticated services and technologies.

As you write your code, you should prefer the use of higher-level frameworks over lower-level frameworks whenever possible. The higher-level frameworks are there to provide object-oriented abstractions for lower-level constructs. These abstractions generally make it much easier to write code because they reduce the number of lines of code you have to write and encapsulate potentially complex features, such as sockets and threads. Although they abstract out lower-level technologies, they do not mask those technologies from you. The lower-level frameworks are still available for developers who prefer using them or who want to use aspects of those frameworks that are not exposed at the higher level.

The following sections provide more detail about what is in each of the exposed layers of iPhone OS, starting with the topmost layers and working downward.

Cocoa Touch Layer

Cocoa Touch is one of the most important layers in iPhone OS. It comprises the key frameworks that provide the infrastructure you need to implement applications in iPhone OS. When developing your applications, you should always start with these frameworks and drop down to lower-level frameworks only as needed.

Apple Push Notification Service

In iPhone OS 3.0 and later, the Apple Push Notification Service provides a way to alert your users of new information, even when your application is not actively running. Using this service, you can push text notifications, trigger audible alerts, or add a numbered badge to your application icon. These messages let users know that they should open your application to receive the related information.
From a design standpoint, there are two parts to making push notifications work for your iPhone applications. First, you need to request the delivery of notifications to your iPhone application and then you need to configure your application delegate to process them. The delegate works together with the shared UIApplication object to perform both of these tasks. Second, you need to provide a server-side process to generate the notifications in the first place. This process lives on your own local server and works with Apple Push Notification Service to trigger the notifications.

For more information about how to configure your application to use remote notifications, see Apple Push Notification Service Programming Guide.

Address Book UI Framework

The Address Book UI framework (AddressBookUI.framework) is an Objective-C programming interface that you use to display standard system interfaces for creating new contacts and for editing and selecting existing contacts. This framework simplifies the work needed to display contact information in your application and also ensures that your application uses the same interfaces as other applications, thus ensuring consistency across the platform.

For more information about the classes of the Address Book UI framework and how to use them, see Address Book Programming Guide for iPhone OS and Address Book UI Framework Reference.

In App Email

In iPhone OS 3.0 and later, the Message UI framework (MessageUI.framework) provides support for composing and queuing email messages in the user's outbox. The composition support consists of a view controller interface that you can present in your application. You can populate the fields of this interface with the contents of the message you want to send. You can set the recipients, subject, body content, and any attachments you want to include with the message. The user then has the option of editing the message prior to accepting it. Once accepted, the message is queued in the user's outbox for delivery.

For more information about the classes of the Message UI framework, see Message UI Framework Reference.

Map Kit Framework

In iPhone OS 3.0 and later, the Map Kit framework (MapKit.framework) provides a map interface that you can embed into your own application. Based on the behavior of this interface within the Maps application, this interface provides a scrollable map view that can be annotated with custom information. You can embed this view inside of your own application views and programmatically set various attributes of the map, including the currently displayed map region and the user's location. You can also define custom annotations or use standard annotations (such as a pin marker) to highlight regions of the map and display additional information.

For more information about the classes of the Map Kit framework, see MapKit Framework Reference.
Peer to Peer Support

In iPhone OS 3.0 and later, the Game Kit framework (GameKit.framework) lets you add peer-to-peer network capabilities to your applications. Specifically, this framework provides support for peer-to-peer connectivity and in-game voice features. Although these features are most commonly found in multiplayer network games, you can incorporate them into non-game applications as well. The framework provides you with networking features through a simple (yet powerful) set of classes built on top of Bonjour. These classes abstract out many of the network details, making it easy for developers who might be inexperienced with networking programming to incorporate networking features into their applications.

For more information about how to use the Game Kit framework, see Game Kit Programming Guide and Game Kit Framework Reference.

UIKit Framework

The UIKit framework (UIKit.framework) contains Objective-C programming interfaces that provide the key infrastructure for implementing graphical, event-driven applications in iPhone OS. Every application in iPhone OS uses this framework to implement its core set of features:

- Application management
- Cut, copy, and paste support
- Graphics and windowing support
- Support for handling touch and motion-based events
- User interface management
- Objects representing the standard system views and controls
- Support for text and web content
- Integration with other applications on the system through URL schemes
- Support for the Apple push notification service; see “Apple Push Notification Service” (page 17)
- Accessibility support for disabled users

In addition to providing the fundamental code for building your application, UIKit also incorporates support for some device-specific features, such as the following:

- Accelerometer data
- The built-in camera (where present)
- The user’s photo library
- Device name and model information
- Battery state information
- Proximity sensor information

For information about the classes of the UIKit framework, see UIKit Framework Reference.
Media Layer

In the Media layer are the graphics, audio, and video technologies geared toward creating the best multimedia experience available on a mobile device. More importantly, these technologies were designed to make it easy for you to build applications that look and sound great. The high-level frameworks in iPhone OS make it easy to create advanced graphics and animations quickly, and the low-level frameworks provide you with access to the tools you need to do things exactly the way you want.

Graphics Technologies

High-quality graphics are an important part of all iPhone applications. Whenever possible, you should use prerendered images together with the existing views and functions of the UIKit framework to draw your application's user interface. There may be situations, however, where the basic classes and functions provided by UIKit are insufficient for your needs. In those situations, you can use the technologies described in the following sections to perform any custom drawing.

Quartz

The Core Graphics framework (CoreGraphics.framework) contains the interfaces for the Quartz 2D drawing API. Quartz is the same advanced, vector-based drawing engine that is used in Mac OS X. It provides support for path-based drawing, anti-aliased rendering, gradients, images, colors, coordinate-space transformations, and PDF document creation, display, and parsing. Although the API is C based, it uses object-based abstractions to represent fundamental drawing objects, making it easy to store and reuse your graphics content.

For more information on how to use Quartz to draw content, see Quartz 2D Programming Guide and Core Graphics Framework Reference.

Core Animation

The Quartz Core framework (QuartzCore.framework) contains the Core Animation interfaces. Core Animation is an advanced animation and compositing technology that uses an optimized rendering path to implement complex animations and visual effects. It provides a high-level, Objective-C interface for configuring animations and effects that are then rendered in hardware for performance. Core Animation is integrated into many parts of iPhone OS, including UIKit classes such as UIView, providing animations for many standard system behaviors. You can also use the Objective-C interface in this framework to create custom animations.

For more information on how to use Core Animation in your applications, see Core Animation Programming Guide and Core Animation Reference Collection.

OpenGL ES

The OpenGL ES framework (OpenGLES.framework) provides tools for drawing 2D and 3D content. It is a C-based framework that works closely with the device hardware to provide high frame rates for full-screen game-style applications.
You always use the OpenGL framework in conjunction with the EAGL interfaces. These interfaces are part of the OpenGL ES framework and provide the interface between your OpenGL ES drawing code and the native window objects of your application.

In iPhone OS 3.0 and later, the OpenGL ES framework includes support for both the OpenGL ES 2.0 and the OpenGL ES 1.1 interface specifications. The 2.0 specification provides support for fragment and vertex shaders and is available only on specific iPhone OS–based devices running iPhone OS 3.0 and later. Support for OpenGL ES 1.1 is available on all iPhone OS–based devices and in all versions of iPhone OS.

For information on how to use OpenGL ES in your applications, see OpenGLESProgrammingGuideforiPhoneOS. For reference information, see OpenGLESFrameworkReference.

Audio Technologies

The audio technologies available in iPhone OS are designed to help you provide a rich audio experience for your users. This includes the ability to play back or record high-quality audio and the ability to trigger the vibration feature on devices that support those capabilities.

The audio technologies in iPhone OS support the following audio formats:

- AAC
- Apple Lossless (ALAC)
- A-law
- IMA/ADPCM (IMA4)
- Linear PCM
- µ-law
- DVI/Intel IMA ADPCM
- Microsoft GSM 6.10
- AES3-2003

AV Foundation

In iPhone OS 2.2 and later, the AV Foundation framework (AVFoundation.framework) contains Objective-C classes for playing audio content. You can use this support to play file- or memory-based sounds of any duration. You can play multiple sounds simultaneously and control various playback aspects of each sound. In iPhone OS 3.0 and later, this framework also includes support for recording audio and managing audio session information.

For more information about the classes of the AV Foundation framework, see AVFoundationFrameworkReference.

Core Audio

Native support for audio is provided by the Core Audio family of frameworks, listed in Table 2-1. Core Audio is a C-based interface that supports the manipulation of stereo-based audio. You can use Core Audio in iPhone OS to generate, record, mix, and play audio in your applications. You can also use Core Audio to access the vibrate capability on devices that support it.
Table 2-1  Core Audio frameworks

<table>
<thead>
<tr>
<th>Framework</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreAudio.framework</td>
<td>Defines the audio data types used throughout Core Audio.</td>
</tr>
<tr>
<td>AudioToolbox.framework</td>
<td>Provides playback and recording services for audio files and streams. This framework also provides support for managing audio files and playing system alert sounds.</td>
</tr>
<tr>
<td>AudioUnit.framework</td>
<td>Provides services for using the built-in audio units, which are audio processing modules.</td>
</tr>
</tbody>
</table>

For more information about Core Audio, see Core Audio Overview. For information about how to use the Audio Toolbox framework to play sounds, see Audio Queue Services Programming Guide and Audio Toolbox Framework Reference.

OpenAL

In addition to Core Audio, iPhone OS includes support for the Open Audio Library (OpenAL). The OpenAL interface is a cross-platform standard for delivering positional audio in applications. You can use it to implement high-performance, high-quality audio in games and other programs that require positional audio output. Because OpenAL is a cross-platform standard, the code modules you write using OpenAL on iPhone OS can be ported to run on many other platforms.

For information about OpenAL, including how to use it, see http://www.openal.org.

Video Technologies

iPhone OS provides support for full-screen video playback through the Media Player framework (MediaPlayer.framework). This framework supports the playback of movie files with the .mov, .mp4, .m4v, and .3gp filename extensions and using the following compression standards:

- H.264 video, up to 1.5 Mbps, 640 by 480 pixels, 30 frames per second, Low-Complexity version of the H.264 Baseline Profile with AAC-LC audio up to 160 Kbps, 48kHz, stereo audio in .m4v, .mp4, and .mov file formats
- H.264 video, up to 768 Kbps, 320 by 240 pixels, 30 frames per second, Baseline Profile up to Level 1.3 with AAC-LC audio up to 160 Kbps, 48kHz, stereo audio in .m4v, .mp4, and .mov file formats
- MPEG-4 video, up to 2.5 Mbps, 640 by 480 pixels, 30 frames per second, Simple Profile with AAC-LC audio up to 160 Kbps, 48kHz, stereo audio in .m4v, .mp4, and .mov file formats
- Numerous audio formats, including the ones listed in “Audio Technologies” (page 21)

For information about the classes of the Media Player framework, see Media Player Framework Reference.
Core Services Layer

Core Services provides the fundamental system services that all applications use. Even if you do not use these services directly, many parts of the system are built on top of them.

Address Book

The Address Book framework (AddressBook.framework) provides programmatic access to the contacts stored on a user's device. If your application uses contact information, you can use this framework to access and modify the records in the user's contacts database. For example, a chat program might use this framework to retrieve the list of possible contacts with which to initiate a chat session and display those contacts in a custom view.

For information about the functions in the Address Book framework, see Address Book Framework Reference.

Core Data

In iPhone OS 3.0 and later, the Core Data framework (CoreData.framework) is a technology for managing the data model of a Model-View-Controller application. Core Data is intended for use in applications where the data model is already highly structured. Instead of defining data structures programmatically, you use the graphical tools in Xcode to build a schema representing your data model. At runtime, instances of your data-model entities are created, managed, and made available through the Core Data framework.

By managing your application's data model for you, Core Data significantly reduces the amount of code you have to write for your application. Core Data also provides the following features:

- Storage of object data in a SQLite database for optimal performance
- A new NSFetchedResultsController class to manage results for table views
- Management of undo/redo beyond basic text editing
- Support for the validation of property values
- Support for propagating changes and ensuring that the relationships between objects remain consistent
- Support for grouping, filtering, and organizing data in memory

If you are starting to develop a new application or are planning a significant update to an existing application, you should consider using Core Data. For an example of how to use Core Data in an iPhone application, see Core Data Tutorial for iPhone OS. For more information about the classes of the Core Data framework, see Core Data Framework Reference.

Core Foundation

The Core Foundation framework (CoreFoundation.framework) is a set of C-based interfaces that provide basic data management and service features for iPhone applications. This framework includes support for the following:

- Collection data types (arrays, sets, and so on)
Bundles
String management
Date and time management
Raw data block management
Preferences management
URL and stream manipulation
Threads and run loops
Port and socket communication

The Core Foundation framework is closely related to the Foundation framework, which provides Objective-C interfaces for the same basic features. When you need to mix Foundation objects and Core Foundation types, you can take advantage of the “toll-free bridging” that exists between the two frameworks. **Toll-free bridging** means that you can use some Core Foundation and Foundation types interchangeably in the methods and functions of either framework. This support is available for many of the data types, including the collection and string data types. The class and type descriptions for each framework state whether an object is toll-free bridged and, if so, what object it is bridged with.

For more information about this framework, see *Core Foundation Framework Reference*.  

### Core Location

The **Core Location framework** (*CoreLocation.framework*) lets you determine the current latitude and longitude of a device. The framework uses the available hardware to triangulate the user’s position based on nearby GPS, cell, or WiFi signal information. The Maps application uses this feature to show the user’s current position on a map. You can incorporate this technology into your own applications to provide position-based information to the user. For example, you might have a service that searches for nearby restaurants, shops, or facilities, and base that search on the user’s current location.

In iPhone OS 3.0 and later, this framework also includes support for accessing compass information on iPhone OS–based devices that include suitable hardware.

For information about the classes of the Core Location framework, see *Core Location Framework Reference*.  

### Foundation Framework

The **Foundation framework** (*Foundation.framework*) provides Objective-C wrappers to many of the features found in the Core Foundation framework, which is described in “Core Foundation” (page 23). The Foundation framework provides support for the following features:

- Collection data types (arrays, sets, and so on)
- Bundles
- String management
- Date and time management
- Raw data block management
In App Purchase

In iPhone OS 3.0 and later, the Store Kit framework (StoreKit.framework) provides support for the purchasing of content and services from within your iPhone applications. For example, you could use this feature to allow the user to unlock additional application features. Or if you are a game developer, you could use it to offer additional game levels. In both cases, the Store Kit framework handles the financial aspects of the transaction, processing payment requests through the user's iTunes Store account and providing your application with information about the purchase.

The Store Kit focuses on the financial aspects of a transaction, ensuring that transactions occur securely and correctly. Your application handles the other aspects of the transaction, including the presentation of a purchasing interface and the downloading (or unlocking) of the appropriate content. This division of labor gives you control over the user experience for purchasing content. You decide what kind of purchasing interface you want to present to the user and when to do so. You also decide on the delivery mechanism that works best for your application.

For information about how to use the Store Kit framework, see In App Purchase Programming Guide and Store Kit Framework Reference.

SQLite

The SQLite library lets you embed a lightweight SQL database into your application without running a separate remote database server process. From your application, you can create local database files and manage the tables and records in those files. The library is designed for general purpose use but is still optimized to provide fast access to database records.

The header file for accessing the SQLite library is located in $iPhoneSDK$/usr/include/sqlite3.h, where $iPhoneSDK$ is the path to the target SDK in your Xcode installation directory. For more information about using SQLite, go to http://www.sqlite.org.

XML Support

The Foundation framework provides the NSXMLParser class for retrieving elements from an XML document. Additional support for manipulating XML content is provided by the libXML2 libraries. This open source library lets you parse or write arbitrary XML data quickly and transform XML content to HTML.
The header files for accessing the `libXML2` library are located in the `<iPhoneSDK>/usr/include/libxml2/` directory, where `<iPhoneSDK>` is the path to the target SDK in your Xcode installation directory. For more information about using `libXML2`, go to http://xmlsoft.org/index.html.

### Core OS Layer

#### CFNetwork

The **CFNetwork framework** (`CFNetwork.framework`) is a set of high-performance, C-based interfaces that provide object-oriented abstractions for working with network protocols. These abstractions give you detailed control over the protocol stack and make it easy to use lower-level constructs such as BSD sockets. You can use this framework to simplify tasks such as communicating with FTP and HTTP servers or resolving DNS hosts. Here are some of the tasks you can perform with the CFNetwork framework. You can:

- Use BSD sockets
- Create encrypted connections using SSL or TLS
- Resolve DNS hosts
- Work with HTTP, authenticating HTTP, and HTTPS servers
- Work with FTP servers
- Publish, resolve, and browse Bonjour services

CFNetwork is based, both physically and theoretically, on BSD sockets. For information on how to use CFNetwork, see [CFNetwork Programming Guide](http://developer.apple.com) and [CFNetwork Framework Reference](http://developer.apple.com).

#### Accessory Support

In iPhone OS 3.0 and later, the **External Accessory framework** (`ExternalAccessory.framework`) provides support for communicating with hardware accessories attached to an iPhone or iPod touch device. Accessories can be connected through the 30-pin dock connector of a device or wirelessly using Bluetooth. The External Accessory framework provides a way for you to get information about each available accessory and to initiate communications sessions. After that, you are free to manipulate the accessory directly using any commands it supports.

Security

In addition to its built-in security features, iPhone OS also provides an explicit Security framework (Security.framework) that you can use to guarantee the security of the data your application manages. This framework provides interfaces for managing certificates, public and private keys, and trust policies. It supports the generation of cryptographically secure pseudo random numbers. It also supports the storage of certificates and cryptographic keys in the keychain, which is a secure repository for sensitive user data.

The Common Crypto interfaces provide additional support for symmetric encryption, HMAC, and Digests. The Digests feature provides functions that are essentially compatible with the functionality normally found in the OpenSSL library, which is not available in iPhone OS.

In iPhone OS 3.0 and later, it is possible for you to share Keychain items among multiple applications you create. Sharing items makes it easier for applications in the same suite to interoperate more smoothly. For example, you could use this feature to share user passwords or other elements that might otherwise require you to prompt the user from each application separately. To share data between applications, you must configure the Xcode project of each application with the proper entitlements.

For information about the functions and features associated with the Security framework, see Security Framework Reference. For information about how to access the Keychain, see Keychain Services Programming Guide. For information about setting up entitlements in your Xcode projects, see iPhone Development Guide. For information about the entitlements you can configure, see the description for the SecItemAdd function in Keychain Services Reference.

System

The system level encompasses the kernel environment, drivers, and low-level UNIX interfaces of the operating system. The kernel itself is based on Mach and is responsible for every aspect of the operating system. It manages the virtual memory system, threads, file system, network, and interprocess communication. The drivers at this layer also provide the interface between the available hardware and system frameworks. However, access to the kernel and drivers is restricted to a limited set of system frameworks and applications for security purposes.

iPhone OS provides a set of interfaces for accessing many low-level features of the operating system. Your application accesses these features through the LibSystem library. The interfaces are C-based and provide support for the following:

- Threading (POSIX threads)
- Networking (BSD sockets)
- File-system access
- Standard I/O
- Bonjour and DNS services
- Locale information
- Memory allocation
- Math computations
Header files for many Core OS technologies are located in the `<iPhoneSDK>/usr/include/` directory, where `<iPhoneSDK>` is the path to the target SDK in your Xcode installation directory. For information about the functions associated with these technologies, see *iPhone OS Manual Pages*. 
Migrating from Cocoa

If you are an existing Cocoa developer, many of the frameworks available in iPhone OS should seem familiar to you. The basic technology stack in iPhone OS is identical in many respects to the one found in Mac OS X. Despite the similarities, however, the frameworks in iPhone OS are not exactly the same as their Mac OS X counterparts. This chapter describes the differences you may encounter as you create iPhone applications and explains how you can adjust to some of the more significant differences.

Note: This chapter is intended for developers who are already familiar with Cocoa terminology and programming techniques. If you want to learn more about the basic design patterns used for Cocoa applications (and iPhone applications), see Cocoa Fundamentals Guide.

General Migration Notes

If your Cocoa application is already factored using the Model-View-Controller design pattern, it should be relatively easy to migrate key portions of your application to iPhone OS. For information about designing applications for iPhone OS, see iPhone Application Programming Guide.

Migrating Your Data Model

Cocoa applications whose data model is based on classes in the Foundation and Core Foundation frameworks can be brought over to iPhone OS with little or no modification. Both frameworks are supported in iPhone OS and are virtually identical to their Mac OS X counterparts, although there are some differences. However, most of the differences are relatively minor or are related to features that would need to be removed in the iPhone version of your application anyway. For example, iPhone applications do not support AppleScript. For a detailed list of differences, see “Foundation Framework Differences” (page 33).

If your Cocoa application is built on top of Core Data, you can migrate that data model to an iPhone application in iPhone OS 3.0 and later; Core Data is not supported in earlier versions of iPhone OS. The Core Data framework in iPhone OS supports binary and SQLite data stores (not XML data stores) and supports migration from existing Cocoa applications. For the supported data stores, you can copy your Core Data resource files to your iPhone application project and use them as is. For information on how to use Core Data in your Xcode projects, see Core Data Programming Guide.

If your Cocoa application displays lots of data on the screen, you might want to simplify your data model when migrating it to iPhone OS. Although you can create rich applications with lots of data in iPhone OS, keep in mind that doing so may not serve your users’ needs. Mobile users typically want only the most important information, in the least amount of time. Providing the user with too much data all at once can be impractical, because of the limited screen space, and may also slow down your application, because of the extra work required to load that data. Refactoring your Cocoa application’s data structures might be worthwhile if it provides better performance and a better user experience in iPhone OS.
Migrating Your User Interface

The user interface in iPhone OS is structured and implemented very differently from the one in Mac OS X. Take, for example, the objects that represent views and windows in Cocoa. Although iPhone OS and Cocoa both have objects representing views and windows, the way those objects work is slightly different on each platform. In addition, you must be more selective about what you display in your views because screen size is limited and your views must be large enough to provide an adequate target for a user’s finger.

In addition to differences in the view objects themselves, there are also significant differences in how you display those views at runtime. For example, if you want to display a lot of data in a Cocoa application, you might increase the window size, use multiple windows, or use tab views to manage that data. In iPhone applications, there is only one window whose size is fixed, so applications must break information into reasonably sized chunks and present those chunks on different sets of views. The goal of chunking information is to create one or more “screens’ worth of content,” which you can use as the basis for defining your views. For example, to display a hierarchical list of data in Cocoa, you could use a single `NSBrowser` object, but in iPhone OS you would need to create distinct sets of views to display the information at each level of the hierarchy. This makes your interface design somewhat more complex, but because it is such a crucial way of displaying information, iPhone provides a considerable amount of support for this type of organization.

View controllers were introduced to Cocoa in Mac OS X v10.5 and may not be in common use yet. In iPhone applications, view controllers provide a critical part of the infrastructure for managing your user interface. View controllers manage the presentation of your user interface. They also work with the system to make sure your application’s resources do not tie up too much memory and degrade performance. Understanding the role of view controllers and how you use them in your application is therefore critical to the design of your user interface.

For general information about the user interface design principles of iPhone OS, see *iPhone Human Interface Guidelines*. For additional information about the windows and views you use to build your interface, and the underlying architecture on which they are built, see *iPhone Application Programming Guide*. For information about view controllers and how you use them to construct the flow of your user interface, see *ViewController Programming Guide for iPhone OS*.

Memory Management

In iPhone OS, you always use the memory-managed model to retain, release, and autorelease objects. Garbage collection is not supported in iPhone OS.

Because memory is more tightly constrained for iPhone OS–based devices than for Macintosh computers, you also need to adjust your use of autorelease pools to prevent the buildup of autoreleased objects. Whenever possible, you should release objects directly rather than autorelease them. When you allocate many objects in a tight loop, you either need to release those objects directly or create autorelease pools at appropriate places in your loop code to free up autoreleased objects at regular intervals. Waiting until the end of your loop could result in a low-memory warning or the termination of your application.

Framework Differences

Although most of the iPhone OS frameworks are also present in Mac OS X, there are platform differences in how those frameworks are implemented and used. The following sections call out some of the key differences that existing Mac OS X developers might notice as they develop iPhone applications.
UIKit Versus AppKit

In iPhone OS, the UIKit framework provides the infrastructure for building graphical applications, managing the event loop, and performing other interface-related tasks. The UIKit framework is completely distinct from the AppKit framework, however, and should be treated as such when designing your iPhone applications. For this reason, when migrating a Cocoa application to iPhone OS, you must replace a significant number of interface-related classes and logic. Table 3-1 lists some of the specific differences between the frameworks to help you understand what is required of your application in iPhone OS.

### Table 3-1 Differences in interface technologies

<table>
<thead>
<tr>
<th>Difference</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document support</td>
<td>In iPhone OS, applications typically have only one window and do not use separate document objects or document windows. Therefore, any files opened by the application are managed by the application directly and used to update the content of its window.</td>
</tr>
<tr>
<td>View classes</td>
<td>UIKit provides a very focused set of custom views and controls for you to use. Many of the views and controls found in AppKit would simply not work well on iPhone and iPod touch devices. Other views have more iPhone-specific alternatives. For example, instead of the NSBrowser class, iPhone uses an entirely different paradigm (navigation controllers) to manage the display of hierarchical information. For a description of the views and controls available in iPhone OS along with information on how to use them, see iPhone Human Interface Guidelines.</td>
</tr>
<tr>
<td>View coordinate systems</td>
<td>In iPhone OS, the drawing model for Quartz and UIKit content is nearly identical to the model in Mac OS X, with one exception. The Mac OS X drawing model uses a coordinate system where the origin for windows and views is in the lower-left corner by default, with axes extending up and to the right. In iPhone OS, the default origin point is in the top-left corner and the axes extend down and to the right. In Mac OS X, this coordinate system is known as a “flipped” coordinate system, but in iPhone OS it is the default coordinate system. For more information about graphics and coordinate systems, see iPhone Application Programming Guide.</td>
</tr>
<tr>
<td>Windows as views</td>
<td>Conceptually, windows and views represent the same constructs in iPhone OS as they do in Mac OS X. In implementation terms, however, the two platforms implement windows and views quite differently. In Mac OS X, the NSWindow class is a subclass of NSResponder, but in iPhone OS, the UIWindow class is actually a subclass of UIView instead. This change in inheritance means that windows use Core Animation layers to implement their drawing surface. The main reason for having window objects at all in UIKit is to support the layering of windows within the operating system. For example, the system displays the status bar in a separate window that floats above your application's window. Another difference between iPhone OS and Mac OS X relates to the use of windows. Whereas a Mac OS X application can have any number of windows, most iPhone applications have only one. Displaying different screens of information in an iPhone application is done by swapping out custom views from the application window rather than by changing the window.</td>
</tr>
<tr>
<td>Difference</td>
<td>Discussion</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Event handling</td>
<td>The UIKit event-handling model is significantly different from the one found in Mac OS X. Instead of mouse and keyboard events, UIKit delivers touch events to your views. These events require you to implement a different set of methods but also require you to make some changes to your overall event-handling code. For example, you would never track a touch event by extracting queued events from a local tracking loop. For more information about handling events in iPhone applications, see <em>iPhone Application Programming Guide</em>.</td>
</tr>
<tr>
<td>Target-action model</td>
<td>UIKit supports three variant forms for action methods, as opposed to just one for AppKit. Controls in UIKit can invoke actions for different phases of the interaction and they have more than one target assigned to the same interaction. Thus, in UIKit a control can deliver multiple distinct actions to multiple targets over the course of a single interaction cycle. For more information about the target-action model in iPhone applications, see <em>iPhone Application Programming Guide</em>.</td>
</tr>
<tr>
<td>Drawing and printing support</td>
<td>The drawing capabilities of UIKit are scaled to support the rendering needs of the UIKit classes. This support includes image loading and display, string display, color management, font management, and a handful of functions for rendering rectangles and getting the graphics context. UIKit does not include a general purpose set of drawing classes because several other alternatives (namely, Quartz and OpenGL ES) are already present in iPhone OS. Printing is not supported because there is no direct support for connecting printers or other print-related hardware to an iPhone or iPod touch. For more information about graphics and drawing, see <em>iPhone Application Programming Guide</em>.</td>
</tr>
<tr>
<td>Text support</td>
<td>The text support in iPhone OS is geared toward composing email and notes. The UIKit classes let applications display and edit simple strings and somewhat more complex HTML content. The more advanced text-layout and glyph-generation features that you would find in an advanced word processor are not included in iPhone OS because they are less relevant for the mobile environment.</td>
</tr>
<tr>
<td>The use of accessor methods versus properties</td>
<td>UIKit makes extensive use of properties throughout its class declarations. Properties were introduced to Mac OS X in version 10.5 and thus came along after the creation of many classes in the AppKit framework. Rather than simply mimic the same getter and setter methods in AppKit, properties are used in UIKit as a way to simplify the class interfaces. For information about how to use properties, see Declared Properties in <em>The Objective-C Programming Language</em>.</td>
</tr>
<tr>
<td>Controls and cells</td>
<td>Controls in UIKit do not use cells. Cells are used in Mac OS X as a lightweight alternative to views. Because views in UIKit are themselves very lightweight objects, cells are not needed. Despite the naming conventions, the cells designed for use with the <code>UITableView</code> class are actually based on the <code>UIView</code> class.</td>
</tr>
<tr>
<td>Table views</td>
<td>The <code>UITableView</code> class in iPhone OS can be thought of as a cross between the <code>NSTableView</code> and <code>NSOutlineView</code> classes in the AppKit framework. It uses features from both of those AppKit classes to create a more appropriate tool for displaying data on a smaller screen. The <code>UITableView</code> class displays a single column at a time and allows you to group related rows together into sections. It is also a means for displaying and editing hierarchical lists of information. For more information about the <code>UITableView</code> class, see <em>UITableView Class Reference</em>.</td>
</tr>
</tbody>
</table>
Discussion

Nearly all applications written for iPhone OS have a much smaller command set than does a comparable Mac OS X application, and so menus are not supported in iPhone OS and are generally unnecessary anyway. For those few commands that are needed, a toolbar or set of buttons is usually more appropriate. For data-based menus, a picker or navigation controller interface is often more appropriate.

Core Animation layers

In iPhone OS, every drawing surface is backed by a Core Animation layer and implicit animation support is provided for many view-related properties. Because of the built-in animation support, you usually do not need to use Core Animation layers explicitly in your code. Most animations can be performed simply by changing the desired property of the affected view. The only time you might need to use layers directly is when you need precise control over the layer tree or when you need features not exposed at the view level. For information about how Core Animation layers are integrated into the drawing model of iPhone OS, see *iPhone Application Programming Guide*.

For information about the classes of UIKit, see *UIKit Framework Reference*.

**Foundation Framework Differences**

A version of the Foundation framework is available in both Mac OS X and iPhone OS, and most of the classes you would expect to be present are available in both. Both frameworks provide support for managing values, strings, collections, threads, and many other common types of data. Table 3-2 lists some of the major areas of functionality that are not included in iPhone OS, however, along with the reasons why the related classes are not available. Wherever possible, this table lists alternative technologies that you can use instead.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metadata and predicate management</td>
<td>Spotlight metadata and search predicates are not supported in iPhone OS because Spotlight itself is not supported.</td>
</tr>
<tr>
<td>Distributed objects and port name server management</td>
<td>The Distributed Objects technology is not available, but you can still use the <code>NSPort</code> family of classes to interact with ports and sockets. You can also use the Core Foundation and CFNetwork frameworks to handle your networking needs.</td>
</tr>
<tr>
<td>Cocoa bindings</td>
<td>Cocoa bindings are not supported in iPhone OS. Instead, iPhone OS uses a slightly modified version of the target-action model that adds flexibility in how you handle actions in your code.</td>
</tr>
<tr>
<td>Objective-C garbage collection</td>
<td>Garbage collection is not supported in iPhone OS. Instead, you must use the memory-managed model, whereby you retain objects to claim ownership and release objects when you no longer need them.</td>
</tr>
<tr>
<td>AppleScript support</td>
<td>AppleScript is not supported in iPhone OS.</td>
</tr>
</tbody>
</table>
The Foundation framework provides support for XML parsing through the **NSXMLParser** class. However, other XML parsing classes (including **NSXMLDocument**, **NSXMLNode**, and **NSXMLElement**) are not available in iPhone OS. In addition to the **NSXMLParser** class, you can also use the **libXML2** library, which provides a C-based XML parsing interface.

For a list of the specific classes that are available in Mac OS X but not in iPhone OS, see the class hierarchy diagram located in “The Foundation Framework” in *Foundation Framework Reference*.

### Changes to Other Frameworks

Table 3-3 lists the key differences in other frameworks found in iPhone OS.

**Table 3-3** Differences in frameworks common to iPhone OS and Mac OS X

<table>
<thead>
<tr>
<th>Framework</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressBook.framework</td>
<td>This framework contains the interfaces for accessing user contacts. In iPhone OS, the interfaces of this framework are written in C instead of Objective-C. For more information, see <em>Address Book Framework Reference</em>.</td>
</tr>
<tr>
<td>AudioToolbox.framework, AudioUnit.framework, CoreAudio.framework</td>
<td>The iPhone OS versions of these frameworks provide support primarily for recording, playing, and mixing of single and multichannel audio content. More advanced audio processing features and custom audio unit plug-ins are not supported. One addition for iPhone OS, however, is the ability to trigger the vibrate option for iPhone devices. For information on how to use the audio support, see Multimedia Support in <em>iPhone Application Programming Guide</em>.</td>
</tr>
<tr>
<td>CFNetwork.framework</td>
<td>This framework contains the Core Foundation Network interfaces. In iPhone OS, the CFNetwork framework is a top-level framework and not a subframework. Most of the actual interfaces remain unchanged, however. For more information, see <em>CFNetwork Framework Reference</em>.</td>
</tr>
<tr>
<td>CoreGraphics.framework</td>
<td>This framework contains the Quartz interfaces. In iPhone OS, the Core Graphics framework is a top-level framework and not a subframework. You can use Quartz to create paths, gradients, shadings, patterns, colors, images, and bitmaps in exactly the same way you do in Mac OS X. There are a few Quartz features that are not present in iPhone OS, however, including PostScript support, image sources and destinations, Quartz Display Services support, and Quartz Event Services support. For more information, see <em>Core Graphics Framework Reference</em>.</td>
</tr>
</tbody>
</table>
### Framework Differences

<table>
<thead>
<tr>
<th>Framework</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenGL ES.framework</td>
<td>OpenGL ES is a version of OpenGL designed specifically for embedded systems. If you are an existing OpenGL developer, the OpenGL ES interface should be familiar to you. However, the OpenGL ES interface still differs in several significant ways. First, it is a much more compact interface, supporting only those features that can be performed efficiently using the available graphics hardware. Second, many of the extensions you might normally use in desktop OpenGL might not be available to you in OpenGL ES. Despite these differences, you should still be able to perform most of the same operations you would normally on the desktop. If you are migrating existing OpenGL code, however, you may have to rewrite some parts of your code to use different rendering techniques in iPhone OS. For information about the OpenGL ES support in iPhone OS, see <em>OpenGLES Programming Guide for iPhone OS</em>.</td>
</tr>
<tr>
<td>QuartzCore.framework</td>
<td>This framework contains the Core Animation interfaces. Most of the Core Animation interfaces are the same for both iPhone OS and Mac OS X. However, in iPhone OS, the classes for managing layout constraints and support for using Core Image filters are not available. In addition, the interfaces for Core Image and Core Video (which are also part of the Mac OS X version of the framework) are not available. For more information, see <em>Quartz Core Framework Reference</em>.</td>
</tr>
<tr>
<td>Security.framework</td>
<td>This framework contains the security interfaces. In iPhone OS, this framework focuses on securing your application data by providing support for encryption and decryption, pseudo-random number generation, and the Keychain. The framework does not contain authentication or authorization interfaces and has no support for displaying the contents of certificates. In addition, the Keychain interfaces are a simplified version of the ones used in Mac OS X. For information about the security support, see <em>iPhone Application Programming Guide</em>.</td>
</tr>
<tr>
<td>System-Configuration.framework</td>
<td>This framework contains networking-related interfaces. In iPhone OS, this framework contains only the reachability interfaces. You use these interfaces to determine how a device is connected to the network, such as whether it’s connected using EDGE, GPRS, or Wi-Fi.</td>
</tr>
</tbody>
</table>
iPhone OS Developer Tools

To develop applications for iPhone OS, you need a Mac OS X computer running the Xcode tools. Xcode is Apple’s suite of development tools that provide support for project management, code editing, building executables, source-level debugging, source-code repository management, performance tuning, and much more. At the center of this suite is the Xcode application itself, which provides the basic source-code development environment. Xcode is not the only tool you use though, and the following sections provide an introduction to the key applications you use to develop software for iPhone OS.

Xcode

The focus of your development experiences is the Xcode application. Xcode is an integrated development environment (IDE) that provides all of the tools you need to create and manage your iPhone projects and source files, build your code into an executable, and run and debug your code either in iPhone simulator or on a device. Xcode incorporates a number of features to make developing iPhone applications easier, including the following:

- A project management system for defining software products
- A code-editing environment that includes features such as syntax coloring, code completion, and symbol indexing
- An advanced documentation viewer for viewing and searching Apple documentation
- A context-sensitive inspector for viewing information about selected code symbols
- An advanced build system with dependency checking and build rule evaluation
- GCC compilers supporting C, C++, Objective-C, Objective-C++, and Objective-C 2.0, and other languages
- Integrated source-level debugging using GDB
- Distributed computing, enabling you to distribute large projects over several networked machines
- Predictive compilation that speeds single-file compile turnaround times
- Advanced debugging features such as fix and continue and custom data formatters
- Advanced refactoring tools that let you make global modifications to your code without changing its overall behavior
- Support for project snapshots, which provide a lightweight form of local source-code management
- Support for launching performance tools to analyze your software
- Support for integrated source-code management
- AppleScript support for automating the build process
- Support for DWARF and Stabs debugging information (DWARF debugging information is generated by default for all new projects)
To create a new iPhone application, you start by creating a new project in Xcode. A project manages all of the information associated with your application, including the source files, build settings, and rules needed to put all of the pieces together. The heart of every Xcode project is the project window, shown in Figure A-1. This window provides quick access to all of the key elements of your application. In the Groups and Files list, you manage the files in your project, including the source files and build targets that are created from those source files. In the toolbar, you access commonly used tools and commands. And in the details pane, you can configure a space for working on your project. Other aspects of the project window provide you with contextual information about your project.

Figure A-1  An Xcode project window

When you build your application in Xcode, you have a choice of building it for iPhone simulator or for a device. The simulator provides a local environment for testing your applications to make sure they behave essentially the way you want. After you are satisfied with your application’s basic behavior, you can tell Xcode to build your application and run it on an iPhone or iPod touch connected to your computer. Running your application on a device provides the ultimate test environment, and Xcode lets you attach the built-in debugger to the code running there.
For details on how to build and run your project on iPhone OS, see *iPhone Development Guide*. For more information about the overall Xcode environment, see *Xcode Overview*.

**Interface Builder**

Interface Builder is the tool you use to assemble your application's user interface visually. Using Interface Builder, you assemble your application's window by dragging and dropping preconfigured components onto it, as shown in Figure A-3. The components include standard system controls such as switches, text fields, and buttons, and also custom views to represent the views your application provides. After you've placed the components on the window's surface, you can position them by dragging them around, configure their attributes using the inspector, and establish the relationships between those objects and your code. When your interface looks the way you want it, you save the contents to a nib file, which is a custom resource file format.
The nib files you create in Interface Builder contain all the information that UIKit needs to recreate the same objects in your application at runtime. Loading a nib file creates runtime versions of all the objects stored in the file, configuring them exactly as they were in Interface Builder. It also uses the connection information you specified to establish connections between the newly created objects and any existing objects in your application. These connections provide your code with pointers to the nib-file objects and also provide the information the objects themselves need to communicate user actions to your code.

Overall, using Interface Builder saves a tremendous amount of time when it comes to creating your application's user interface. Interface Builder eliminates the custom code needed to create, configure, and position the objects that make up your interface. Because it is a visual editor, you get to see exactly what your interface will look like at runtime.

For more information about using Interface Builder, see Interface Builder User Guide.

Instruments

To ensure that you deliver the best user experience for your software, the Instruments environment lets you analyze the performance of your iPhone applications while running in the simulator or on a device. Instruments gathers data from your running application and presents that data in a graphical display called the timeline. You can gather data about your application's memory usage, disk activity, network activity, and graphics performance. The timeline view can display all of the different types of information side by side, letting you correlate the overall behavior of your application, not just the behavior in one specific area. To get even more detailed information, you can also view the detailed samples that Instruments gathers.
In addition to providing the timeline view, Instruments provides tools to help you analyze your application's behavior over time. For example, the Instruments window lets you store data from multiple runs so that you can see whether your application's behavior is actually improving or whether it still needs work. You can save the data from these runs in an Instruments document and open them at any time.

For details on how to use Instruments with iPhone applications, see *iPhone Development Guide*. For general information on how to use Instruments, see *Instruments User Guide*.

**Shark**

Shark is a powerful tool that you can use to analyze the performance of your iPhone applications. Shark lets you profile your code using several different options while it is running on an iPhone OS–based device. The results of profiling are a statistical sampling of your application's runtime behavior that can be viewed and analyzed using the Shark data mining and charting tools. These tools can help you visualize your program's runtime behavior and identify potential hot spots.

For more information about using Shark with iPhone OS–based devices, see *Shark User Guide*. 
APPENDIX A

iPhone OS Developer Tools
## IPhone OS Frameworks

This appendix contains information about the frameworks of iPhone OS. These frameworks provide the interfaces you need to write software for the platform. Where applicable, the tables in this index list any key prefixes used by the classes, methods, functions, types, or constants of the framework. You should avoid using any of the specified prefixes in your own symbol names.

### Device Frameworks

Table B-1 describes the frameworks available for use in iPhone OS–based devices. You can find these frameworks in the `<Xcode>/Platforms/iPhoneOS.platform/Developer/SDKs/<iPhoneSDK>/System/Library/Frameworks` directory where `<Xcode>` is the path to your Xcode installation directory and `<iPhoneSDK>` is the specific SDK version you are targeting. The “First available” column lists the iPhone OS release in which the framework first appeared.

<table>
<thead>
<tr>
<th>Name</th>
<th>First available</th>
<th>Prefixes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressBook.framework</td>
<td>2.0</td>
<td>AB</td>
<td>Contains functions for accessing the user’s contacts database directly. See Address Book Framework Reference.</td>
</tr>
<tr>
<td>AddressBookUI.framework</td>
<td>2.0</td>
<td>AB</td>
<td>Contains classes for displaying the system-defined people picker and editor interfaces. See Address Book UI Framework Reference.</td>
</tr>
<tr>
<td>AudioToolbox.framework</td>
<td>2.0</td>
<td>Audio</td>
<td>Contains the interfaces for handling audio stream data and for playing and recording audio. See Audio Toolbox Framework Reference.</td>
</tr>
<tr>
<td>AudioUnit.framework</td>
<td>2.0</td>
<td>Audio</td>
<td>Contains the interfaces for loading and using audio units. See Audio Unit Framework Reference.</td>
</tr>
<tr>
<td>AVFoundation.framework</td>
<td>2.2</td>
<td>AV</td>
<td>Contains Objective-C interfaces for playing and recording audio. See AV Foundation Framework Reference.</td>
</tr>
<tr>
<td>CFNetwork.framework</td>
<td>2.0</td>
<td>CF</td>
<td>Provides the data types used throughout Core Audio. See CFNetwork Framework Reference.</td>
</tr>
</tbody>
</table>
## Device Frameworks

<table>
<thead>
<tr>
<th>Name</th>
<th>First available</th>
<th>Prefixes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreData.framework</td>
<td>3.0</td>
<td>NS</td>
<td>Contains interfaces for managing your application's data model. See Core Data Framework Reference.</td>
</tr>
<tr>
<td>CoreFoundation.framework</td>
<td>2.0</td>
<td>CF</td>
<td>Provides fundamental software services, including abstractions for common data types, string utilities, collection utilities, resource management, and preferences. See Core Foundation Framework Reference.</td>
</tr>
<tr>
<td>CoreGraphics.framework</td>
<td>2.0</td>
<td>CG</td>
<td>Contains the interfaces for Quartz 2D. See Core Graphics Framework Reference.</td>
</tr>
<tr>
<td>CoreLocation.framework</td>
<td>2.0</td>
<td>CL</td>
<td>Contains the interfaces for determining the user’s location. See Core Location Framework Reference.</td>
</tr>
<tr>
<td>ExternalAccessory.framework</td>
<td>3.0</td>
<td>EA</td>
<td>Contains interfaces for communicating with attached hardware accessories. See External Accessory Framework Reference.</td>
</tr>
<tr>
<td>GameKit.framework</td>
<td>3.0</td>
<td>GK</td>
<td>Contains the interfaces for managing peer-to-peer connectivity. See Game Kit Framework Reference.</td>
</tr>
<tr>
<td>Foundation.framework</td>
<td>2.0</td>
<td>NS</td>
<td>Contains the classes and methods for the Cocoa Foundation layer. See Foundation Framework Reference.</td>
</tr>
<tr>
<td>IOKit.framework</td>
<td>2.0</td>
<td>N/A</td>
<td>Contains interfaces used by the device. Do not include this framework directly.</td>
</tr>
<tr>
<td>MapKit.framework</td>
<td>3.0</td>
<td>MK</td>
<td>Contains classes for embedding a map interface into your application and for looking up reverse geocoding coordinates. See MapKit Framework Reference.</td>
</tr>
<tr>
<td>MediaPlayer.framework</td>
<td>2.0</td>
<td>MP</td>
<td>Contains interfaces for playing full-screen video. See Media Player Framework Reference.</td>
</tr>
<tr>
<td>MessageUI.framework</td>
<td>3.0</td>
<td>MF</td>
<td>Contains interfaces for composing and queuing email messages. See Message UI Framework Reference.</td>
</tr>
<tr>
<td>MobileCoreServices.framework</td>
<td>3.0</td>
<td>UT</td>
<td>Defines the uniform type identifiers (UTIs) supported by the system.</td>
</tr>
<tr>
<td>OpenAL.framework</td>
<td>2.0</td>
<td>AL</td>
<td>Contains the interfaces for OpenAL, a cross-platform positional audio library. For more information, go to <a href="http://www.openal.org">http://www.openal.org</a>.</td>
</tr>
</tbody>
</table>
### iPhone OS Frameworks

<table>
<thead>
<tr>
<th>Name</th>
<th>First available</th>
<th>Prefixes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenGLES.framework</td>
<td>2.0</td>
<td>EAGL, GL</td>
<td>Contains the interfaces for OpenGL ES, which is an embedded version of the OpenGL cross-platform 2D and 3D graphics rendering library. See OpenGL ES Framework Reference.</td>
</tr>
<tr>
<td>QuartzCore.framework</td>
<td>2.0</td>
<td>CA</td>
<td>Contains the Core Animation interfaces. See Quartz Core Framework Reference.</td>
</tr>
<tr>
<td>Security.framework</td>
<td>2.0</td>
<td>CSSM, Sec</td>
<td>Contains interfaces for managing certificates, public and private keys, and trust policies. See Security Framework Reference.</td>
</tr>
<tr>
<td>StoreKit.framework</td>
<td>3.0</td>
<td>SK</td>
<td>Contains interfaces for handling the financial transactions associated with in app purchases. See Store Kit Framework Reference.</td>
</tr>
<tr>
<td>System-Configuration.framework</td>
<td>2.0</td>
<td>SC</td>
<td>Contains interfaces for determining the network configuration of a device. See System Configuration Framework Reference.</td>
</tr>
<tr>
<td>UIKit.framework</td>
<td>2.0</td>
<td>UI</td>
<td>Contains classes and methods for the iPhone application user-interface layer. See UIKit Framework Reference.</td>
</tr>
</tbody>
</table>

### Simulator Frameworks

Although you should always target the device frameworks when writing your code, you might need to compile your code specially for the simulator during testing. The frameworks available on the device and in the simulator are mostly identical, but there are a handful of differences. For example, the simulator uses several Mac OS X frameworks as part of its own implementation. In addition, the exact interfaces available for a device framework and a simulator framework may differ slightly due to system limitations. For a list of frameworks, and for information about the specific differences between the device and simulator frameworks, see iPhone Development Guide.

### System Libraries

Note that some specialty libraries at the Core OS and Core Services level are not packaged as frameworks. Instead, iPhone OS includes many dynamic libraries in the `/usr/lib` directory of the system. Dynamic shared libraries are identified by their `.dylib` extension. Header files for the libraries are located in the `/usr/include` directory.

Each version of the iPhone SDK includes a local copy of the dynamic shared libraries that are installed with the system. These copies are installed on your development system so that you can link to them from your Xcode projects. To see the list of libraries for a particular version of iPhone OS, look in `<Xcode>/Platforms/iPhoneOS.platform/Developer/SDKs/<iPhoneSDK>/usr/lib` where `<Xcode>`
is the path to your Xcode installation directory and <iPhoneSDK> is the specific SDK version you are targeting. For example, the shared libraries for the iPhone OS 3.0 SDK would be located in the
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS3.0.sdk/usr/lib directory, with the corresponding headers in
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS3.0.sdk/usr/include.

iPhone OS uses symbolic links to point to the most current version of most libraries. When linking to a dynamic shared library, use the symbolic link instead of a link to a specific version of the library. Library versions may change in future versions of iPhone OS; if your software is linked to a specific version, that version might not always be available on the user's system.
This table describes the changes to *iPhone OS Technology Overview*.

<table>
<thead>
<tr>
<th>Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-10-19</td>
<td>Added links to reference documentation in framework appendix.</td>
</tr>
<tr>
<td>2009-05-27</td>
<td>Updated for iPhone OS 3.0.</td>
</tr>
<tr>
<td>2008-10-15</td>
<td>New document that introduces iPhone OS and its technologies.</td>
</tr>
</tbody>
</table>