

iOS Technology Overview



Contents

About the iOS Technologies 7

At a Glance 7

The iOS Architecture is Layered 7

The iOS Technologies Are Packaged as Frameworks 8

iOS and OS X Share Many of the Same Frameworks 9

You Use Xcode to Develop Apps 9

The Developer Library is There to Help You 9

How to Use This Document 10

See Also 10

Cocoa Touch Layer 11

High-Level Features 11

Auto Layout 11

Storyboards 12

Document Support 12

Multitasking 12

Printing 13

UI State Preservation 13

Apple Push Notification Service 14

Local Notifications 14

Gesture Recognizers 14

Peer-to-Peer Services 15

Standard System View Controllers 15

External Display Support 16

Cocoa Touch Frameworks 16

Address Book UI Framework 16

Event Kit UI Framework 16

Game Kit Framework 17

iAd Framework 17

Map Kit Framework 18

Message UI Framework 18

Twitter Framework 18

UIKit Framework 19

Media Layer 21

Graphics Technologies 21

Audio Technologies 22

Video Technologies 23

AirPlay 24

Media Layer Frameworks 24

Assets Library Framework 24

AV Foundation Framework 25

Core Audio 25

Core Graphics Framework 26

Core Image Framework 26

Core MIDI Framework 27

Core Text Framework 27

Core Video Framework 27

Image I/O Framework 27

GLKit Framework 28

Media Player Framework 28

OpenAL Framework 29

OpenGL ES Framework 29

Quartz Core Framework 29

Core Services Layer 30

High-Level Features 30

iCloud Storage 30

Automatic Reference Counting 32

Block Objects 32

Data Protection 33

File-Sharing Support 33

Grand Central Dispatch 34

In-App Purchase 34

SQLite 35

XML Support 35

Core Services Frameworks 35

Accounts Framework 35

Address Book Framework 36

Ad Support Framework 36

CFNetwork Framework 36

Core Data Framework 37

Core Foundation Framework 37

Core Location Framework	38
Core Media Framework	38
Core Motion Framework	39
Core Telephony Framework	39
Event Kit Framework	39
Foundation Framework	39
Mobile Core Services Framework	40
Newsstand Kit Framework	40
Pass Kit Framework	41
Quick Look Framework	41
Social Framework	41
Store Kit Framework	42
System Configuration Framework	42

Core OS Layer 43

Accelerate Framework	43
Core Bluetooth Framework	43
External Accessory Framework	44
Generic Security Services Framework	44
Security Framework	44
System	45

Migrating from Cocoa 46

General Migration Notes	46
Migrating Your Data Model	46
Migrating Your User Interface	47
Memory Management	48
Framework Differences	48
UIKit Versus AppKit	48
Foundation Framework Differences	52
Changes to Other Frameworks	52

iOS Developer Tools 55

Xcode	55
Instruments	57
The Developer Library	58

iOS Frameworks 61

Device Frameworks	61
Simulator Frameworks	66

[System Libraries](#) 66

[Document Revision History](#) 68

Figures and Tables

About the iOS Technologies 7

Figure I-1 Layers of iOS 8

Media Layer 21

Table 2-1 Core Audio frameworks 26

Migrating from Cocoa 46

Table 5-1 Differences in interface technologies 48

Table 5-2 Foundation technologies unavailable in iOS 52

Table 5-3 Differences in frameworks common to iOS and OS X 53

iOS Developer Tools 55

Figure A-1 An Xcode project window 56

Figure A-2 Running a project from Xcode 57

Figure A-3 Using Instruments to tune your application 58

Figure A-4 The iOS Developer Library 59

iOS Frameworks 61

Table B-1 Device frameworks 61

About the iOS Technologies

iOS is the operating system that runs on iPhone, iPod touch, and iPad devices. The operating system manages the device hardware and provides the technologies required to implement native apps. The operating system also ships with various system apps, such as Phone, Mail, and Safari, that provide standard system services to the user.

The **iOS Software Development Kit (SDK)** contains the tools and interfaces needed to develop, install, run, and test native apps that appear on an iOS device's Home screen. Native apps are built using the iOS system frameworks and Objective-C language and run directly on iOS. Unlike web apps, native apps are installed physically on a device and are therefore always available to the user, even when the device is in Airplane mode. They reside next to other system apps and both the app and any user data is synced to the user's computer through iTunes.

In addition to native apps, it is possible to create web apps using a combination of HTML, cascading style sheets (CSS), and JavaScript code. Web apps run inside the Safari web browser and require a network connection to access your web server. Native apps, on the other hand, are installed directly on the device and can run without the presence of a network connection.

At a Glance

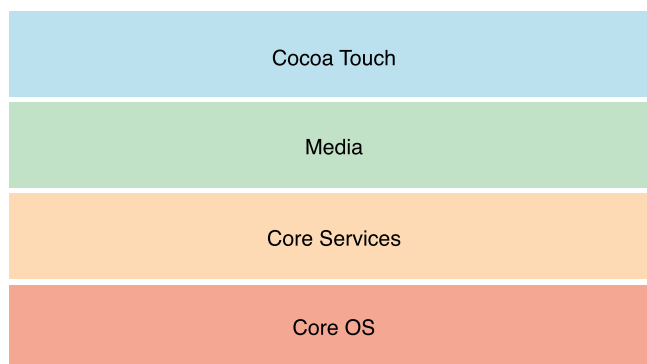
The iOS SDK provides the resources you need to develop native iOS apps. Therefore, understanding a little about the technologies and tools that make up this SDK can help you make better choices about how to design and implement your apps.

The iOS Architecture is Layered

At the highest level, iOS acts as an intermediary between the underlying hardware and the apps that appear on the screen. The apps you create rarely talk to the underlying hardware directly. Instead, apps communicate with the hardware through a set of well-defined system interfaces that protect your app from hardware changes. This abstraction makes it easy to write apps that work consistently on devices with different hardware capabilities.

The implementation of iOS technologies can also be viewed as a set of layers, which are shown in Figure I-1. At the lower layers of the system are the fundamental services and technologies on which all apps rely; higher-level layers contain more sophisticated services and technologies.

Figure I-1 Layers of iOS



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 amount 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 to use them or who want to use aspects of those frameworks that are not exposed by the higher layers.

Relevant chapters: [“Cocoa Touch Layer”](#) (page 11), [“Media Layer”](#) (page 21), [“Core Services Layer”](#) (page 30), [“Core OS Layer”](#) (page 43)

The iOS Technologies Are Packaged as Frameworks

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 apps, and so on) needed to support that library. To use frameworks, you link them into your app 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.

Relevant sections: [“iOS Frameworks”](#) (page 61)

iOS and OS X Share Many of the Same Frameworks

If you are an existing Cocoa developer, writing iOS apps should feel familiar. Many of the technologies found in OS X can also be found in iOS. The biggest differences between the two platforms occur at the user interface level but even then there are similarities in how you present and manage your app’s interface. As a result, porting apps from OS X to iOS is possible with a little bit of work.

Relevant chapter: [“Migrating from Cocoa”](#) (page 46)

You Use Xcode to Develop Apps

Xcode is the development environment you use to create, test, debug, and tune your apps. Xcode comprises the Xcode app, which is a wrapper for all of the other tools you need to build your apps, including Instruments and iOS Simulator. You use Xcode to write your code and then run your apps in iOS Simulator or directly on an attached iOS device. After debugging your app, you can use Xcode to launch Instruments and profile your app’s performance.

Development on an actual device requires signing up for Apple’s paid iOS Developer Program and configuring a device for development purposes. You can obtain a copy of Xcode and the iOS SDK and you find out more about the iOS Developer Program at the [iOS Dev Center](#).

Relevant chapters: [“iOS Developer Tools”](#) (page 55)

The Developer Library is There to Help You

The iOS Developer Library is an important resource for you to use during development. It contains reference information for the various technologies. It also contains programming guides, release notes, tech notes, sample code, and many other resources offering tips and guidance about the best way to create your apps.

You can access the iOS Developer Library from the [Apple Developer website](#) or from Xcode. In Xcode, choosing Help > Developer Documentation displays the Xcode documentation window, which is the central resource for accessing information about iOS development. You can use this window to browse the documentation, perform searches, and bookmark documents you may want to refer to later.

Relevant sections: [“The Developer Library”](#) (page 58)

How to Use This Document

iOS Technology Overview is an introductory guide for anyone who is new to the iOS 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 iOS platform.
- Learn about iOS 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 iOS from other platforms.
- Find key documents relating to the technologies you are interested in.

This document does not provide information about user-level features that have no impact on the software development process, nor does this document list the hardware capabilities of specific iOS devices.

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

See Also

If you are new to iOS development, this book provides only an overview of the system. To learn more about how to develop iOS apps, you should read the following documents:

- *Start Developing iOS Apps Today* provides a guided tour of the development process, starting with how to set up your system and ending with the process of how to submit apps to the App Store. If you are new to developing iOS apps, this is another good starting point for exploring iOS app development.
- *iOS Human Interface Guidelines* provides guidance and information about how to design your app’s user interface.
- *Tools Workflow Guide for iOS* describes the iOS development process from the perspective of the tools. This document covers the configuration and provisioning of devices for development and covers the use of Xcode (and other tools) for building, running, and testing your software.

Cocoa Touch Layer

The **Cocoa Touch** layer contains the key frameworks for building iOS applications. This layer defines the basic application infrastructure and support for key technologies such as multitasking, touch-based input, push notifications, and many high-level system services. When designing your applications, you should investigate the technologies in this layer first to see if they meet your needs.

High-Level Features

The following sections describe some of the key technologies available in the Cocoa Touch layer.

Auto Layout

Introduced in iOS 6, auto layout improves upon the “springs and struts” model previously used to lay out the elements of a user interface. With auto layout, you define rules for how to lay out the elements in your user interface. These rules express a larger class of relationships and are more intuitive to use than springs and struts. For example, you can specify that a button should always be 20 points from the left edge of its parent view.

The entities used in Auto Layout are Objective-C objects called constraints. This approach brings you a number of benefits:

- Localization through swapping of strings alone, instead of also revamping layouts.
- Mirroring of user-interface elements for right-to-left languages like Hebrew and Arabic.
- Better layering of responsibility between objects in the view and controller layers.

A view object usually knows best about its standard size, its positioning within its superview, and its positioning relative to its sibling views. A view controller can override these values if something nonstandard is required.

For more information about using auto layout, see *Cocoa Auto Layout Guide*.

Storyboards

Introduced in iOS 5, storyboards supplant nib files as the recommended way to design your application's user interface. Unlike nib files, storyboards let you design your entire user interface in one place so you can see all of your views and view controllers and how they work together. An important part of storyboards is the ability to define segues, which are transitions from one view controller to another. Applications can define these transitions visually in Xcode or initiate them programmatically in Xcode. These transitions allow you to capture the flow of your user interface in addition to the content.

You can use a single storyboard file to store all of your application's view controllers and views, or you can use multiple view storyboards to organize portions of your interface. At build time, Xcode takes the contents of the storyboard file and divides it up into discrete pieces that can be loaded individually for better performance. Your application never needs to manipulate these pieces directly, though. The UIKit framework provides convenience classes for accessing the contents of a storyboard from your code.

For more information about using storyboards to design your interface, see *Xcode 4 User Guide*. For information about how to access storyboards from your code, see the *UINavigationController Class Reference*.

Document Support

Introduced in iOS 5, the UIKit framework introduced the `UIDocument` class for managing the data associated with user documents. This class makes implementing document-based applications much easier, especially applications that store documents in iCloud. In addition to providing a container for all of your document-related data, the `UIDocument` class provides built-in support for asynchronous reading and writing of file data, safe saving of data, automatic saving of data, support for detecting iCloud conflicts, and support for flat file or package file representations. For applications that use Core Data for their data model, you can use the `UIManagedDocument` subclass to manage your data stores.

For information about using documents in your apps, see *Document-Based App Programming Guide for iOS*.

Multitasking

Applications built using iOS SDK 4.0 or later (and running in iOS 4.0 and later) are not terminated when the user presses the Home button; instead, they shift to a background execution context. The multitasking support defined by UIKit helps your application transition to and from the background state smoothly.

To preserve battery life, most applications are suspended by the system shortly after entering the background. A suspended application remains in memory but does not execute any code. This behavior allows an application to resume quickly when it is relaunched without consuming battery power in the meantime. However, applications may be allowed to continue running in the background for the following reasons:

- An application can request a finite amount of time to complete some important task.

- An application can declare itself as supporting specific services that require regular background execution time.
- An application can use local notifications to generate user alerts at designated times, whether or not the application is running.

Regardless of whether your application is suspended or continues running in the background, supporting multitasking does require additional work on your part. The system notifies your application as it transitions to and from the background. These notification points are your cue to perform any important application tasks such as saving user data.

For more information on how to handle multitasking transitions, and for information on how to request background execution time, see *iOS App Programming Guide*.

Printing

Introduced in iOS 4.2, the UIKit printing support allows applications to send content wirelessly to nearby printers. For the most part, UIKit does all of the heavy lifting associated with printing. It manages the printing interfaces, works with your application to render the printable content, and handles the scheduling and execution of print jobs on the printer.

Print jobs submitted by your application are handed off to the printing system, which manages the actual printing process. Print jobs from all applications on a device are queued and printed on a first-come, first-served basis. Users can get the status of print jobs from the Print Center application and can even use that application to cancel print jobs. All other aspects of printing are handled for you automatically by the system.

Note: Wireless printing is available only on devices that support multitasking. You can use the `UIPrintInteractionController` object to detect whether printing is available to your application.

For information about how to incorporate printing support into your applications, see “Printing” in *Drawing and Printing Guide for iOS*.

UI State Preservation

Introduced in iOS 6, state preservation makes it easier for apps to restore their user interface to the state it was in when the user last used it. When an app moves to the background, it is asked to save the semantic state of its views and view controllers. Upon relaunch, the app uses this state to restore its interface and make it seem as if the app had never quit. Support for state preservation is integrated into UIKit, which provides the infrastructure for saving and restoring your app’s interface.

For more information about how to add state preservation support to your app, see *iOS App Programming Guide*.

Apple Push Notification Service

Introduced in iOS 3.0, the Apple Push Notification Service provides a way to alert users of new information, even when your application is not actively running. Using this service, you can push text notifications, add a badge to your application icon, or trigger audible alerts on user devices at any time. 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 iOS applications. First, the application must request the delivery of notifications and process the notification data once it is delivered. 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 *Local and Push Notification Programming Guide*.

Local Notifications

Introduced in iOS 4.0, local notifications complement the existing push notification mechanism by giving applications an avenue for generating the notifications locally instead of relying on an external server. Applications running in the background can use local notifications as a way to get a user's attention when important events happen. For example, a navigation application running in the background can use local notifications to alert the user when it is time to make a turn. Applications can also schedule the delivery of local notifications for a future date and time and have those notifications delivered even if the application is not running.

The advantage of local notifications is that they are independent of your application. Once a notification is scheduled, the system manages the delivery of it. Your application does not even have to be running when the notification is delivered.

For more information about using local notifications, see *Local and Push Notification Programming Guide*.

Gesture Recognizers

Introduced in iOS 3.2, gesture recognizers are objects that you attach to views and use to detect common types of gestures such as swipes and pinches. After attaching a gesture recognizer to your view, you tell it what action to perform when the gesture occurs. The gesture recognizer object then tracks the raw touch events and applies the system-defined heuristics for what the given gesture should be. Without gesture recognizers, you must do all this work yourself, which can be quite complicated.

UIKit includes a `UIGestureRecognizer` class that defines the basic behavior for all gesture recognizers. You can define your own custom gesture recognizer subclasses or use one of the UIKit-supplied subclasses to handle any of the following standard gestures:

- Tapping (any number of taps)
- Pinching in and out (for zooming)
- Panning or dragging
- Swiping (in any direction)
- Rotating (fingers moving in opposite directions)
- Long presses

For more information about the available gesture recognizers, see *Event Handling Guide for iOS*.

Peer-to-Peer Services

Introduced in iOS 3.0, the Game Kit framework provides peer-to-peer connectivity over Bluetooth. You can use peer-to-peer connectivity to initiate communication sessions with nearby devices and implement many of the features found in multiplayer games. Although primarily used in games, you can also use this feature in other types of applications.

For information about how to use peer-to-peer connectivity features in your application, see *Game Center Programming Guide*. For an overview of the Game Kit framework, see [“Game Kit Framework”](#) (page 17).

Standard System View Controllers

Many of the frameworks in the Cocoa Touch layer contain view controllers for presenting standard system interfaces. You are encouraged to use these view controllers in your applications to present a consistent user experience. Whenever you need to perform one of the following tasks, you should use a view controller from the corresponding framework:

- **Display or edit contact information**—Use the view controllers in the Address Book UI framework.
- **Create or edit calendar events**—Use the view controllers in the Event Kit UI framework.
- **Compose an email or SMS message**—Use the view controllers in the Message UI framework.
- **Open or preview the contents of a file**—Use the `UIDocumentInteractionController` class in the UIKit framework.
- **Take a picture or choose a photo from the user’s photo library**—Use the `UIImagePickerController` class in the UIKit framework.
- **Shoot a video clip**—Use the `UIImagePickerController` class in the UIKit framework.

For information on how to present and dismiss view controllers, see *View Controller Programming Guide for iOS*. For information about the interface presented by a specific view controller, see *View Controller Catalog for iOS*.

External Display Support

Introduced in iOS 3.2, external display support allows some iOS-based devices to be connected to an external display through a set of supported cables. When connected, the associated screen can be used by the application to display content. Information about the screen, including its supported resolutions, is accessible through the interfaces of the UIKit framework. You also use that framework to associate your application's windows with one screen or another.

For more information about how to connect to, and display content on, an external display, see *View Programming Guide for iOS*.

Cocoa Touch Frameworks

The following sections describe the frameworks of the Cocoa Touch layer and the services they offer.

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 iOS* and *Address Book UI Framework Reference for iOS*.

Event Kit UI Framework

Introduced in iOS 4.0, the Event Kit UI framework (`EventKitUI.framework`) provides view controllers for presenting the standard system interfaces for viewing and editing calendar-related events. This framework builds upon the event-related data in the Event Kit framework, which is described in [“Event Kit Framework”](#) (page 39).

For more information about the classes and methods of this framework, see *Event Kit UI Framework Reference*.

Game Kit Framework

Introduced in iOS 3.0, 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 applications other than games as well. The framework provides networking features through a simple (yet powerful) set of classes built on top of Bonjour. These classes abstract out many of the network details. For developers who might be inexperienced with networking programming, the framework allows them to incorporate networking features into their applications.

Introduced in iOS 4.1, Game Center is an extension to the Game Kit framework that provides support for the following features:

- Aliases, to allow users to create their own online persona. Users log in to Game Center and interact with other players anonymously through their alias. Players can set status messages as well as mark specific people as their friends.
- Leaderboards, to allow your application to post user scores to Game Center and retrieve them later. You might use this feature to show the best scores among all users of your application.
- Matchmaking, to allow you to create multiplayer games by connecting players who are logged into Game Center. Players do not have to be local to each other to join a multiplayer game.
- Achievements, to allow you to record the progress a player has made in your game.
- Challenges, allow a player to challenge a friend to beat an achievement or score. (iOS 6 and later)

In iOS 5 and later, you can use the `GKTurnBasedMatch` class to implement support for turn-based gaming, which allows games to create persistent matches whose state is stored in iCloud. Your game manages the state information for the match and determines which player must act to advance the state of the match.

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

iAd Framework

Introduced in iOS 4.0, the iAd framework (`iAd.framework`) lets you deliver banner-based advertisements from your application. Advertisements are incorporated into standard views that you integrate into your user interface and present when you want. The views themselves work with Apple's ad service to automatically handle all the work associated with loading and presenting the ad content and responding to taps in those ads.

For more information about using iAd in your applications, see *iAd Programming Guide* and *iAd Framework Reference*.

Map Kit Framework

Introduced in iOS 3.0, the Map Kit framework (`MapKit.framework`) provides a scrollable map interface that you can integrate into your existing view hierarchies. You can use this map to provide directions or highlight points of interest. Applications can programmatically set attributes of the map or let the user navigate the map freely. You can also annotate the map with custom images or content.

In iOS 4.0, the basic map view gained support for draggable annotations and custom overlays. Draggable annotations allow you to reposition an annotation, either programmatically or through user interactions, after it has been placed on the map. Overlays offer a way to create complex map annotations that comprise more than one point. For example, you can use overlays to layer information such as bus routes, election maps, park boundaries, or weather information (such as radar data) on top of the map.

In iOS 6.0, you can now create a routing app, whose job is to provide directions to users. When the user requests transit-related directions, the Maps app now lets the user choose the app from which to receive those directions. In addition, all apps can ask the Maps app to provide driving directions and display multiple points-of-interest.

For more information about using classes of the Map Kit framework, see *Location Awareness Programming Guide* and *Map Kit Framework Reference*.

Message UI Framework

Introduced in iOS 3.0, 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 present in your application. You can prepopulate the fields of this view controller to set the recipients, subject, body content, and any attachments you want to include with the message. After presenting the view controller, the user then has the option of editing the message prior to sending it.

In iOS 4.0 and later, this framework provides a view controller for presenting an SMS composition screen. You can use this view controller to create and edit SMS messages without leaving your application. As with the mail composition interface, this interface gives the user the option to edit the message before sending it.

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

Twitter Framework

In iOS 6, the Twitter framework (`Twitter.framework`) is replaced by the Social framework, which supports a UI for generating tweets and support for creating URLs to access the Twitter service. For more information about this framework, see [“Social Framework”](#) (page 41).

In iOS 5, you can use the Twitter framework (`Twitter.framework`) to generate Twitter requests on behalf of the user and to composing and sending tweets. For information about the classes of the Twitter framework, see *Twitter Framework Reference*.

UIKit Framework

The UIKit framework (`UIKit.framework`) provides the key infrastructure for implementing graphical, event-driven applications in iOS. Every iOS application uses this framework to implement the following core features:

- Application management
- User interface management, including support for storyboards and nib files
- Graphics and windowing support, including support for multiple displays
- Multitasking support; see [“Multitasking”](#) (page 12)
- Printing support; see [“Printing”](#) (page 13)
- Support for customizing the appearance of standard UIKit controls (iOS 5 and later)
- Support for implementing view controllers that incorporate content from other view controllers (iOS 5 and later)
- Support for handling touch and motion-based events
- Objects representing the standard system views and controls
- Support for text and web content
- Cut, copy, and paste support
- Support for animating user-interface content
- Integration with other applications on the system through URL schemes and framework interfaces
- Accessibility support for disabled users
- Support for the Apple Push Notification Service; see [“Apple Push Notification Service”](#) (page 14)
- Local notification scheduling and delivery; see [“Local Notifications”](#) (page 14)
- PDF creation
- Support for using custom input views that behave like the system keyboard
- Support for creating custom text views that interact with the system keyboard
- Support for sharing content through email, Twitter, Facebook, and other services

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
- Remote-control information from attached headsets

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

Media Layer

The Media layer contains the graphics, audio, and video technologies geared toward creating the best multimedia experience available on a mobile device. The technologies in this layer were designed to make it easy for you to build applications that look and sound great.

Graphics Technologies

High-quality graphics are an important part of all iOS applications. The simplest (and most efficient) way to create an application is to use prerendered images together with the standard views and controls of the UIKit framework and let the system do the drawing. However, there may be situations where you need to go beyond simple graphics. In those situations, you can use the following technologies to manage your application's graphical content:

- Core Graphics (also known as Quartz) handles native 2D vector- and image-based rendering; see [“Core Graphics Framework”](#) (page 26).
- Core Animation (part of the Quartz Core framework) provides advanced support for animating views and other content; see [“Quartz Core Framework”](#) (page 29).
- Core Image provides advanced support for manipulating video and still images; see [“Core Image Framework”](#) (page 26).
- OpenGL ES and GLKit provide support for 2D and 3D rendering using hardware-accelerated interfaces; see [“OpenGL ES Framework”](#) (page 29) and [“GLKit Framework”](#) (page 28).
- Core Text provides a sophisticated text layout and rendering engine; see [“Core Text Framework”](#) (page 27).
- Image I/O provides interfaces for reading and writing most image formats; see [“Image I/O Framework”](#) (page 27).
- The Assets Library framework provides access to the photos and videos in the user's photo library; see [“Assets Library Framework”](#) (page 24).

For the most part, applications running on devices with Retina displays should work with little or no modifications. Any content you draw is automatically scaled as needed to support high-resolution screens. For vector-based drawing code, the system frameworks automatically use any extra pixels to improve the crispness

of your content. And if you use images in your application, UIKit provides support for loading high-resolution variants of your existing images automatically. For more information about what you need to do to support high-resolution screens, see “App-Related Resources” in *iOS App Programming Guide*.

For information about the graphics-related frameworks, see the corresponding entries in “[Media Layer Frameworks](#)” (page 24).

Audio Technologies

The audio technologies available in iOS are designed to help you provide a rich audio experience for your users. This experience includes the ability to play high-quality audio, record high-quality audio, and trigger the vibration feature on certain devices.

The system provides several ways to play back and record audio content. The frameworks in the following list are ordered from high level to low level, with the Media Player framework offering the highest-level interfaces you can use. When choosing an audio technology, remember that higher-level frameworks are easier to use and are generally preferred. Lower-level frameworks offer more flexibility and control but require you to do more work.

- The Media Player framework provides easy access to the user’s iTunes library and support for playing tracks and playlists; see “[Media Player Framework](#)” (page 28).
- The AV Foundation framework provides a set of easy-to-use Objective-C interfaces for managing audio playback and recording; see “[AV Foundation Framework](#)” (page 25).
- OpenAL provides a set of cross-platform interfaces for delivering positional audio; see “[OpenAL Framework](#)” (page 29).
- The Core Audio frameworks offer both simple and sophisticated interfaces for playing and recording audio content. You use these interfaces for playing system alert sounds, triggering the vibrate capability of a device, and managing the buffering and playback of multichannel local or streamed audio content; see “[Core Audio](#)” (page 25).

The audio technologies in iOS 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

For information about each of the audio frameworks, see the corresponding entry in [“Media Layer Frameworks”](#) (page 24).

Video Technologies

Whether you are playing movie files from your application or streaming them from the network, iOS provides several technologies to play your video-based content. On devices with the appropriate video hardware, you can also use these technologies to capture video and incorporate it into your application.

The system provides several ways to play and record video content that you can choose depending on your needs. When choosing a video technology, remember that the higher-level frameworks simplify the work you have to do to support the features you need and are generally preferred. The frameworks in the following list are ordered from highest to lowest level, with the Media Player framework offering the highest-level interfaces you can use.

- The `UIImagePickerController` class in UIKit provides a standard interface for recording video on devices with a supported camera.
- The Media Player framework provides a set of simple-to-use interfaces for presenting full- or partial-screen movies from your application; see [“Media Player Framework”](#) (page 28).
- The AV Foundation framework provides a set of Objective-C interfaces for managing the capture and playback of movies; see [“AV Foundation Framework”](#) (page 25).
- Core Media describes the low-level data types used by the higher-level frameworks and provides low-level interfaces for manipulating media; see [“Core Media Framework”](#) (page 38).

The video technologies in iOS support 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, 48 kHz, 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, 48 kHz, 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, 48 kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats

- Numerous audio formats, including the ones listed in [“Audio Technologies”](#) (page 22)

For information about each of the video frameworks in the Media layer, see the corresponding entry in [“Media Layer Frameworks”](#) (page 24). For more information on using the `UIImagePickerController` class, see *Camera Programming Topics for iOS*.

AirPlay

AirPlay is a technology that lets your application stream audio to Apple TV and to third-party AirPlay speakers and receivers. AirPlay support is built in to the AV Foundation framework and the Core Audio family of frameworks. Any audio content you play using these frameworks is automatically made eligible for AirPlay distribution. Once the user chooses to play your audio using AirPlay, it is routed automatically by the system.

In iOS 5, users can mirror the content of an iPad 2 to an Apple TV 2 using AirPlay for any application. And developers who want to display different content (instead of mirroring) can assign a new window object to any `UIScreen` objects connected to an iPad 2 via AirPlay. iOS 5 also offers more ways to deliver content over AirPlay, including using the `AVPlayer` class in the AV Foundation framework and the `UIWebView` class in the UIKit framework. In addition, the Media Player framework now includes support for displaying “Now Playing” information in several places, including as part of the content delivered over AirPlay.

For information on how to take advantage of AirPlay, see *AirPlay Overview*.

Media Layer Frameworks

The following sections describe the frameworks of the Media layer and the services they offer.

Assets Library Framework

Introduced in iOS 4.0, the Assets Library framework (`AssetsLibrary.framework`) provides a query-based interface for retrieving photos and videos from the user’s device. Using this framework, you can access the same assets that are normally managed by the Photos application, including items in the user’s saved photos album and any photos and videos that were imported onto the device. You can also save new photos and videos back to the user’s saved photos album.

For more information about the classes and methods of this framework, see *Assets Library Framework Reference*.

AV Foundation Framework

Introduced in iOS 2.2, the AV Foundation framework (`AVFoundation.framework`) contains Objective-C classes for playing audio content. You can use these classes 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 iOS 3.0 and later, this framework also includes support for recording audio and managing audio session information.

In iOS 4.0 and later, the services offered by this framework were expanded to include:

- Media asset management
- Media editing
- Movie capture
- Movie playback
- Track management
- Metadata management for media items
- Stereophonic panning
- Precise synchronization between sounds
- An Objective-C interface for determining details about sound files, such as the data format, sample rate, and number of channels

In iOS 5, the AV Foundation framework includes support for streaming audio and video content over AirPlay using the `AVPlayer` class. AirPlay support is enabled by default, but applications can opt out as needed.

The AV Foundation framework is a single source for recording and playing back audio and video in iOS. This framework also provides much more sophisticated support for handling and managing media items than higher-level frameworks.

For more information about the classes of the AV Foundation framework, see *AV Foundation Framework Reference*.

Core Audio

Native support for audio is provided by the Core Audio family of frameworks, which are 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 iOS to generate, record, mix, and play audio in your applications. You can also use Core Audio to trigger the vibrate capability on devices that support it.

Table 2-1 Core Audio frameworks

Framework	Services
CoreAudio.framework	Defines the audio data types used throughout Core Audio. <i>Core Audio Framework Reference</i> .
AudioToolbox.framework	Provides playback and recording services for audio files and streams. This framework also provides support for managing audio files, playing system alert sounds, and triggering the vibrate capability on some devices. See <i>Audio Toolbox Framework Reference</i> .
AudioUnit.framework	Provides services for using the built-in audio units, which are audio processing modules. <i>Audio Unit Framework Reference</i> .
CoreMIDI.framework	Provides low-level MIDI services. See <i>Core MIDI Framework Reference</i> .
MediaToolbox.framework	Provides access to the audio tap interfaces.

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*.

Core Graphics Framework

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 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 Image Framework

Introduced in iOS 5, the Core Image framework (CoreImage.framework) provides a powerful set of built-in filters for manipulating video and still images. You can use the built-in filters for everything from simple operations (like touching up and correcting photos) to more advanced operations (like face and feature detection). The advantage of using these filters is that they operate in a nondestructive manner so that your original images are never changed directly. In addition, Core Image takes advantage of the available CPU and GPU processing power to ensure that operations are fast and efficient.

The `CIIImage` class provides access to a standard set of filters that you can use to improve the quality of a photograph. To create other types of filters, you can create and configure a `CIFilter` object for the appropriate filter type.

For information about the classes and filters of the Core Image framework, see *Core Image Reference Collection*.

Core MIDI Framework

Introduced in iOS 4.2, the Core MIDI framework (`CoreMIDI.framework`) provides a standard way to communicate with MIDI devices, including hardware keyboards and synthesizers. You use this framework to send and receive MIDI messages and to interact with MIDI peripherals connected to an iOS-based device using the dock connector or network.

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

Core Text Framework

Introduced in iOS 3.2, the Core Text framework (`CoreText.framework`) contains a set of simple, high-performance C-based interfaces for laying out text and handling fonts. The Core Text framework provides a complete text layout engine that you can use to manage the placement of text on the screen. The text you manage can also be styled with different fonts and rendering attributes.

This framework is intended for use by applications that require sophisticated text handling capabilities, such as word-processing applications. If your application requires only simple text input and display, you should continue to use the existing text classes of the UIKit framework.

For more information about using the Core Text interfaces, see *Core Text Programming Guide* and *Core Text Reference Collection*.

Core Video Framework

Introduced in iOS 4.0, the Core Video framework (`CoreVideo.framework`) provides buffer and buffer pool support for the Core Media framework (described in “[Core Media Framework](#)” (page 38)). Most applications never need to use this framework directly.

Image I/O Framework

Introduced in iOS 4.0, the Image I/O framework (`ImageIO.framework`) provides interfaces for importing and exporting image data and image metadata. This framework makes use of the Core Graphics data types and functions and supports all of the standard image types available in iOS.

In iOS 6 and later, you can use this framework to access EXIF and IPTC metadata properties for images.

For more information about the functions and data types of this framework, see *Image I/O Reference Collection*.

GLKit Framework

Introduced in iOS 5, the GLKit framework (`GLKit.framework`) contains a set of Objective-C based utility classes that simplify the effort required to create an OpenGL ES 2.0 application. GLKit provides support for four key areas of application development:

- The `GLKView` and `GLKViewController` classes provide a standard implementation of an OpenGL ES-enabled view and associated rendering loop. The view manages the underlying framebuffer object on behalf of the application; your application just draws to it.
- The `GLKTextureLoader` class provides image conversion and loading routines to your application, allowing it to automatically load texture images into your context. It can load textures synchronously or asynchronously. When loading textures asynchronously, your application provides a completion handler block to be called when the texture is loaded into your context.
- The GLKit framework provides implementations of vector, matrix, and quaternions as well as a matrix stack operation that provides the same functionality found in OpenGL ES 1.1.
- The `GLKBaseEffect`, `GLKSkyboxEffect`, and `GLKReflectionMapEffect` classes provide existing, configurable graphics shaders that implement commonly used graphics operations. In particular, the `GLKBaseEffect` class implements the lighting and material model found in the OpenGL ES 1.1 specification, simplifying the effort required to migrate an application from OpenGL ES 1.1 to OpenGL ES 2.0.

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

Media Player Framework

The Media Player framework (`MediaPlayer.framework`) provides high-level support for playing audio and video content from your application. You can use this framework to play video using a standard system interface.

In iOS 3.0, support was added for accessing the user's iTunes music library. With this support, you can play music tracks and playlists, search for songs, and present a media picker interface to the user.

In iOS 3.2, changes were made to support the playback of video from a resizable view. (Previously, only full-screen support was available.) In addition, numerous interfaces were added to support the configuration and management of movie playback.

In iOS 5, support was added for displaying "Now Playing" information in the lock screen and multitasking controls. This information can also be displayed on an Apple TV and with content delivered via AirPlay. There are also interfaces for detecting whether video is being streamed over AirPlay.

For information about the classes of the Media Player framework, see *Media Player Framework Reference*. For information on how to use these classes to access the user's iTunes library, see *iPod Library Access Programming Guide*.

OpenAL Framework

The Open Audio Library (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 iOS can be ported to many other platforms easily.

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

OpenGL ES Framework

The OpenGL ES framework (OpenGL ES 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 defined by UIKit.

In iOS 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 iOS-based devices running iOS 3.0 and later. Support for OpenGL ES 1.1 is available on all iOS-based devices and in all versions of iOS.

For information on how to use OpenGL ES in your applications, see *OpenGL ES Programming Guide for iOS*. For reference information, see *OpenGL ES Framework Reference*.

Quartz Core Framework

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 iOS, 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*.

Core Services Layer

The Core Services layer contains 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.

High-Level Features

The following sections describe some of the key technologies available in the Core Services layer.

iCloud Storage

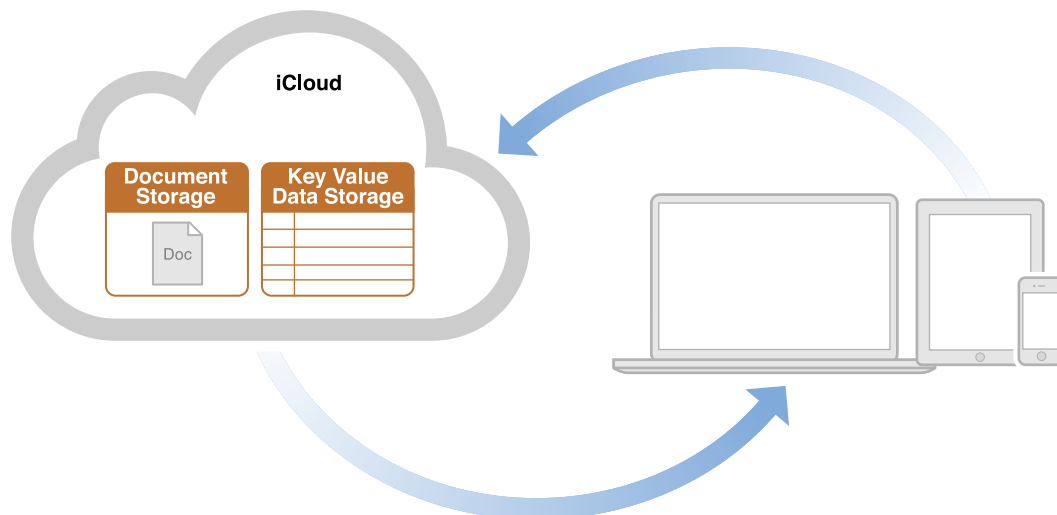
Introduced in iOS 5, iCloud storage lets your application write user documents and data to a central location and access those items from all of a user's computers and iOS devices. Making a user's documents ubiquitous using iCloud means that a user can view or edit those documents from any device without having to sync or transfer files explicitly. Storing documents in a user's iCloud account also provides a layer of safety for that user. Even if a user loses a device, the documents on that device are not lost if they are in iCloud storage.

There are two ways that applications can take advantage of iCloud storage, each of which has a different intended usage:

- **iCloud document storage**—Use this feature to store user documents and data in the user's iCloud account.
- **iCloud key-value data storage**—Use this feature to share small amounts of data among instances of your application.

Most applications will use iCloud document storage to share documents from a user's iCloud account. This is the feature that users think of when they think of iCloud storage. A user cares about whether documents are shared across devices and can see and manage those documents from a given device. In contrast, the iCloud

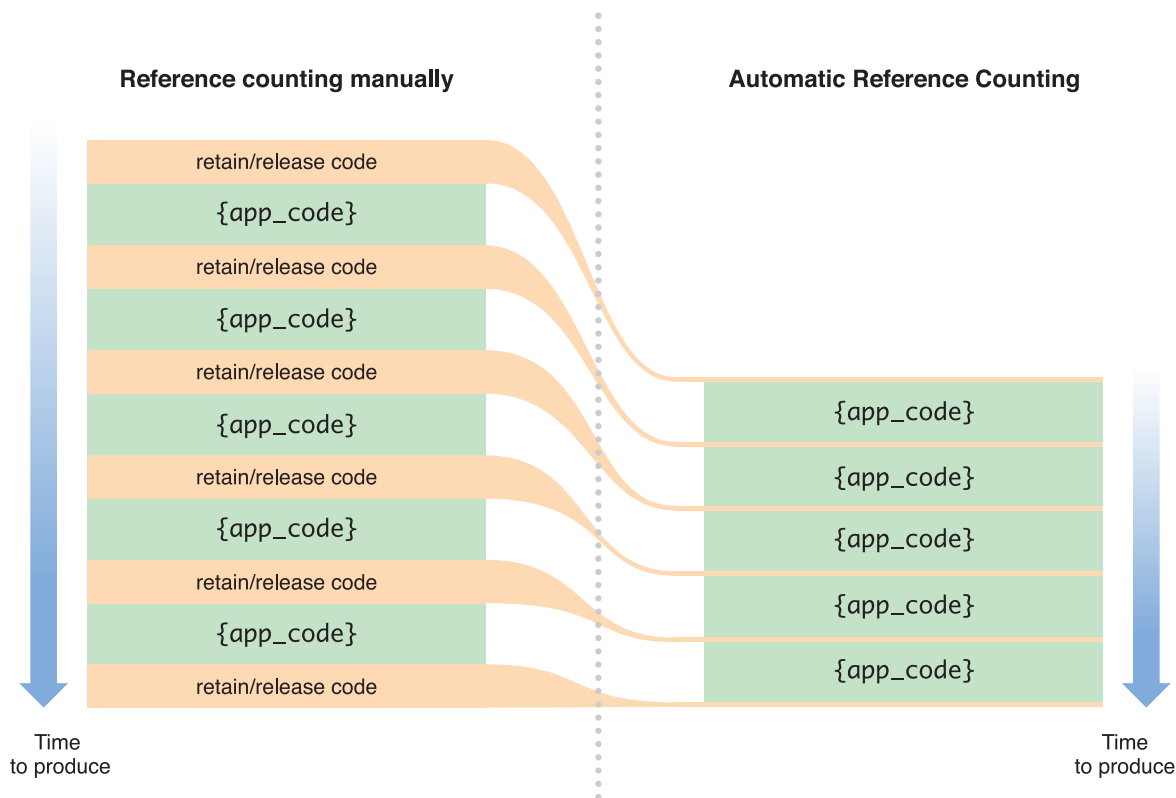
key-value data store is not something a user would see. It is a way for your application to share very small amounts of data (tens of kilobytes) with other instances of itself. Applications should use this feature to store noncritical application data, such as preferences, rather than important application data.



For an overview of how you incorporate iCloud support into your application, see *iCloud Design Guide*.

Automatic Reference Counting

Introduced in iOS 5, **Automatic Reference Counting (ARC)** is a compiler-level feature that simplifies the process of managing the lifetimes of Objective-C objects. Instead of you having to remember when to retain or release an object, ARC evaluates the lifetime requirements of your objects and automatically inserts the appropriate method calls at compile time.



ARC replaces the traditional managed memory model style of programming found in earlier versions of iOS. Any new projects you create automatically use ARC. And Xcode provides migration tools to help convert existing projects to use ARC. For more information about ARC itself, see *Transitioning to ARC Release Notes*.

Block Objects

Introduced in iOS 4.0, block objects are a C-level language construct that you can incorporate into your C and Objective-C code. A block object is essentially an anonymous function and the data that goes with that function, something which in other languages is sometimes called a *closure* or *lambda*. Blocks are particularly useful as callbacks or in places where you need a way of easily combining both the code to be executed and the associated data.

In iOS, blocks are commonly used in the following scenarios:

- As a replacement for delegates and delegate methods
- As a replacement for callback functions
- To implement completion handlers for one-time operations
- To facilitate performing a task on all the items in a collection
- Together with dispatch queues, to perform asynchronous tasks

For an introduction to block objects and their uses, see *A Short Practical Guide to Blocks*. For more information about blocks, see *Blocks Programming Topics*.

Data Protection

Introduced in iOS 4.0, data protection allows applications that work with sensitive user data to take advantage of the built-in encryption available on some devices. When your application designates a specific file as protected, the system stores that file on-disk in an encrypted format. While the device is locked, the contents of the file are inaccessible to both your application and to any potential intruders. However, when the device is unlocked by the user, a decryption key is created to allow your application to access the file.

In iOS 5 and later, data protection includes additional security levels for protected files. These levels let you access an open file even if the user locks the device and access a file after device boot and first unlock, even if the user subsequently locks the device again.

Implementing data protection requires you to be considerate in how you create and manage the data you want to protect. Applications must be designed to secure the data at creation time and to be prepared for access changes when the user locks and unlocks the device.

For more information about how to add data protection to the files of your application, see *iOS App Programming Guide*.

File-Sharing Support

Introduced in iOS 3.2, file-sharing support lets applications make user data files available via iTunes 9.1 and later. An application that declares its support for file sharing makes the contents of its `/Documents` directory available to the user. The user can then move files in and out of this directory as needed from iTunes. This feature does not allow your application to share files with other applications on the same device; that behavior requires the pasteboard or a document interaction controller object.

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

1. Add the `UIFileSharingEnabled` key to your application's `Info.plist` file and set the value of the key to YES.
2. Put whatever files you want to share in your application's `Documents` directory.

3. When the device is plugged into the user's computer, iTunes displays a File Sharing section in the Apps tab of the selected device.
4. The user can add files to this directory or move files to the desktop.

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

For additional information about the `UIFileSharingEnabled` key, see *Information Property List Key Reference*.

Grand Central Dispatch

Introduced in iOS 4.0, Grand Central Dispatch (GCD) is a BSD-level technology that you use to manage the execution of tasks in your application. GCD combines an asynchronous programming model with a highly optimized core to provide a convenient (and more efficient) alternative to threading. GCD also provides convenient alternatives for many types of low-level tasks, such as reading and writing file descriptors, implementing timers, and monitoring signals and process events.

For more information about how to use GCD in your applications, see *Concurrency Programming Guide*. For information about specific GCD functions, see *Grand Central Dispatch (GCD) Reference*.

In-App Purchase

Introduced in iOS 3.0, in-app purchase gives you the ability to vend content and services from inside your application. This feature is implemented using the Store Kit framework, which provides the infrastructure needed to process financial transactions using the user's iTunes account. Your application handles the overall user experience and the presentation of the content or services available for purchase.

In iOS 6, support was added for content hosting and for purchasing iTunes content from inside your app. With content hosting, you can now store your downloadable content on Apple's servers. With iTunes purchase, your app identifies the items you want to make available for purchase but the rest of the transaction is handled for you by Store Kit.

For more information about supporting in-app purchase, see *In-App Purchase Programming Guide*. For additional information about the Store Kit framework, see ["Store Kit Framework"](#) (page 42).

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 `<iOS_SDK>/usr/include/sqlite3.h`, where `<iOS_SDK>` 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 library. 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 `<iOS_SDK>/usr/include/libxml2/` directory, where `<iOS_SDK>` 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 Services Frameworks

The following sections describe the frameworks of the Core Services layer and the services they offer.

Accounts Framework

Introduced in iOS 5, the Accounts framework (`Accounts.framework`) provides a single sign-on model for certain user accounts. Single sign-on improves the user experience, because applications no longer need to prompt a user separately for login information related to an account. It also simplifies the development model for you by managing the account authorization process for your application. In iOS 5.0, applications can use this framework in conjunction with the Twitter framework to access a user's Twitter account.

For more information about the classes of the Accounts framework, see *Accounts Framework Reference*.

Address Book Framework

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.

In iOS 6 and later, access to a user's contacts requires explicit permission from the user. Apps must therefore be prepared for the user to deny that access. Apps are also encouraged to provide `Info.plist` keys describing the reason for needing access.

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

Ad Support Framework

Introduced in iOS 6, the Ad Support framework (`AdSupport.framework`) provides access to an identifier that apps can use for advertising purposes. This framework also provides a flag that indicates whether the user has opted out of ad tracking. Apps are required to read and honor the opt-out flag before trying to access the advertising identifier.

For more information about this framework, see *Ad Support Framework Reference*.

CFNetwork Framework

The CFNetwork framework (`CFNetwork.framework`) is a set of high-performance C-based interfaces that use 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. With the CFNetwork framework, you can:

- Use BSD sockets
- Create encrypted connections using SSL or TLS
- Resolve DNS hosts
- Work with HTTP servers, authenticating HTTP servers, 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* and *CFNetwork Framework Reference*.

Core Data Framework

Introduced in iOS 3.0, 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 in which 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
- An `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 iOS application, see *Core Data Tutorial for iOS*. For more information about the classes of the Core Data framework, see *Core Data Framework Reference*.

Core Foundation Framework

The Core Foundation framework (`CoreFoundation.framework`) is a set of C-based interfaces that provide basic data management and service features for iOS 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 connected to.

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

Core Location Framework

The Core Location framework (`CoreLocation.framework`) provides location and heading information to applications. For location information, the framework uses the onboard GPS, cell, or Wi-Fi radios to find the user’s current longitude and latitude. 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 iOS 3.0, support was added for accessing compass-based heading information on iOS-based devices that include a magnetometer.

In iOS 4.0, support was introduced for a low-power location-monitoring service that uses cellular towers to track changes in the user’s location.

For information about how to use Core Location to gather location and heading information, see *Location Awareness Programming Guide* and *Core Location Framework Reference*.

Core Media Framework

Introduced in iOS 4.0, the Core Media framework (`CoreMedia.framework`) provides the low-level media types used by the AV Foundation framework. Most applications never need to use this framework, but it is provided for those few developers who need more precise control over the creation and presentation of audio and video content.

For more information about the functions and data types of this framework, see *Core Media Framework Reference*.

Core Motion Framework

The Core Motion framework (`CoreMotion.framework`) provides a single set of interfaces for accessing all motion-based data available on a device. The framework supports accessing both raw and processed accelerometer data using a new set of block-based interfaces. For devices with a built-in gyroscope, you can retrieve the raw gyro data as well as processed data reflecting the attitude and rotation rates of the device. You can use both the accelerometer and gyro-based data for games or other applications that use motion as input or as a way to enhance the overall user experience.

For more information about the classes and methods of this framework, see *Core Motion Framework Reference*.

Core Telephony Framework

Introduced in iOS 4.0, the Core Telephony framework (`CoreTelephony.framework`) provides interfaces for interacting with phone-based information on devices that have a cellular radio. Applications can use this framework to get information about a user's cellular service provider. Applications interested in cellular call events (such as VoIP applications) can also be notified when those events occur.

For more information about using the classes and methods of this framework, see *Core Telephony Framework Reference*.

Event Kit Framework

Introduced in iOS 4.0, the Event Kit framework (`EventKit.framework`) provides an interface for accessing calendar events on a user's device. You can use this framework to get existing events and add new events to the user's calendar. Calendar events can include alarms that you can configure with rules for when they should be delivered.

In iOS 6, support was added for creating and accessing reminders on the user's device. The reminders you create show up in the Reminders app along with ones created by the user. In addition, access to a calendar and reminder data requires explicit permission from the user. Apps must therefore be prepared for the user to deny that access. Apps are also encouraged to provide `Info.plist` keys describing the reason for needing access.

For more information about the classes and methods of this framework, see *Event Kit Framework Reference*. See also [“Event Kit UI Framework”](#) (page 16).

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 Framework”](#) (page 37). 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
- Preferences management
- URL and stream manipulation
- Threads and run loops
- Bonjour
- Communication port management
- Internationalization
- Regular expression matching
- Cache support

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

Mobile Core Services Framework

Introduced in iOS 3.0, the Mobile Core Services framework (`MobileCoreServices.framework`) defines the low-level types used in Uniform Type Identifiers (UTIs).

For more information about the types defined by this framework, see *Uniform Type Identifiers Reference*.

Newsstand Kit Framework

Introduced in iOS 5, Newsstand provides a central place for users to read magazines and newspapers. Publishers who want to deliver their magazine and newspaper content through Newsstand can create their own iOS applications using the Newsstand Kit framework (`NewsstandKit.framework`), which lets you initiate background downloads of new magazine and newspaper issues. After you start a download, the system handles the download operation and notifies your application when the new content is available.

For information about the classes you use to manage Newsstand downloads, see *Newsstand Kit Framework Reference*. For information about how to use push notifications to notify your applications, see *Local and Push Notification Programming Guide*.

Pass Kit Framework

Introduced in iOS 6, Pass Kit uses web services, a new file format, and an Objective-C framework (`PassKit.framework`) to implement support for downloadable passes. Companies can create passes to represent items such as coupons, boarding passes, event tickets, and discount cards for businesses. Instead of carrying a physical representation of these items, users can now store them on their iOS device and use them the same way as before.

Passes are created by your company's web service and delivered to the user's device via email, Safari, or your custom app. The pass itself uses a special file format and is cryptographically signed before being delivered. The file format identifies relevant information about the service being offered so that the user knows what it is for. It might also contain a bar code or other information that you can then use to validate the card so that it can be redeemed or used.

For more information about Pass Kit and for information how to add support for it into your apps, see *Passbook Programming Guide*.

Quick Look Framework

Introduced in iOS 4.0, the Quick Look framework (`QuickLook.framework`) provides a direct interface for previewing the contents of files your application does not support directly. This framework is intended primarily for applications that download files from the network or that otherwise work with files from unknown sources. After obtaining the file, you use the view controller provided by this framework to display the contents of that file directly in your user interface.

For more information about the classes and methods of this framework, see *Quick Look Framework Reference for iOS*.

Social Framework

Introduced in iOS 6, the Social framework (`Social.framework`) provides a simple interface for accessing the user's social media accounts. This framework supplants the Twitter framework that was introduced in iOS 5 and adds support for other social accounts, including Facebook and Sina's Weibo service. Apps can use this framework to post status updates and images to a user's account. This framework works with the Accounts framework to provide a single sign-on model for the user and to ensure that access to the user's account is approved.

For more information about the Social framework, see *Social Framework Reference*.

Store Kit Framework

Introduced in iOS 3, the Store Kit framework (`StoreKit.framework`) provides support for the purchasing of content and services from within your iOS 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*.

System Configuration Framework

The System Configuration framework (`SystemConfiguration.framework`) provides the reachability interfaces, which you can use to determine the network configuration of a device. You can use this framework to determine whether a Wi-Fi or cellular connection is in use and whether a particular host server can be accessed.

For more information about the interfaces of this framework, see *System Configuration Framework Reference*. For an example of how to use this framework to obtain network information, see the *Reachability* sample code project.

Core OS Layer

The Core OS layer contains the low-level features that most other technologies are built upon. Even if you do not use these technologies directly in your applications, they are most likely being used by other frameworks. And in situations where you need to explicitly deal with security or communicating with an external hardware accessory, you do so using the frameworks in this layer.

Accelerate Framework

Introduced in iOS 4.0, the Accelerate framework (`Accelerate.framework`) contains interfaces for performing DSP, linear algebra, and image-processing calculations. The advantage of using this framework over writing your own versions of these interfaces is that they are optimized for all of the hardware configurations present in iOS-based devices. Therefore, you can write your code once and be assured that it runs efficiently on all devices.

For more information about the functions of the Accelerate framework, see *Accelerate Framework Reference*.

Core Bluetooth Framework

The Core Bluetooth framework (`CoreBluetooth.framework`) allows developers to interact specifically with Bluetooth Low-Energy ("LE") accessories. The Objective-C interfaces of this framework allow you to scan for LE accessories, connect and disconnect to ones you find, read and write attributes within a service, register for service and attribute change notifications, and much more.

For more information about the interfaces of the Core Bluetooth framework, see *Core Bluetooth Framework Reference*.

External Accessory Framework

Introduced in iOS 3.0, the External Accessory framework (`ExternalAccessory.framework`) provides support for communicating with hardware accessories attached to an iOS-based 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.

For more information about how to use this framework, see *External Accessory Programming Topics*. For information about developing accessories for iOS-based devices, go to <http://developer.apple.com>.

Generic Security Services Framework

Introduced in iOS 5, the Generic Security Services framework (`GSS.framework`) provides a standard set of security-related services to iOS applications. The basic interfaces of this framework are specified in IETF [RFC 2743](#) and [RFC 4401](#). In addition to offering the standard interfaces, iOS includes some additions for managing credentials that are not specified by the standard but that are required by many applications.

For information about the interfaces of the GSS framework, see the header files.

Security Framework

In addition to its built-in security features, iOS 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 pseudorandom 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 library provides additional support for symmetric encryption, HMAC, and digests. The digests feature provides functions that are essentially compatible with those in the OpenSSL library, which is not available in iOS.

In iOS 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 *Tools Workflow Guide for iOS*. 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. For security purposes, access to the kernel and drivers is restricted to a limited set of system frameworks and applications.

iOS 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 `<iOS_SDK>/usr/include/` directory, where `<iOS_SDK>` is the path to the target SDK in your Xcode installation directory. For information about the functions associated with these technologies, see *iOS Manual Pages*.

Migrating from Cocoa

If you are a Cocoa developer, many of the frameworks available in iOS should already seem familiar to you. The basic technology stack in iOS is identical in many respects to the one in OS X. Despite the similarities, however, the frameworks in iOS are not exactly the same as their OS X counterparts. This chapter describes the differences you may encounter as you create iOS 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 iOS 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 iOS.

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 iOS with little or no modification. Both frameworks are supported in iOS and are virtually identical to their OS X counterparts. Most of the differences that do exist are relatively minor or are related to features that would need to be removed in the iOS version of your application anyway. For example, iOS applications do not support AppleScript. For a detailed list of differences, see [“Foundation Framework Differences”](#) (page 52).

If your Cocoa application is built on top of Core Data, you can migrate that data model to an iOS application in iOS 3.0 and later; Core Data is not supported in earlier versions of iOS. The Core Data framework in iOS 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 iOS 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 iOS. Although you can create rich applications with lots of data in iOS, 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 more limited screen space, and it could 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 iOS.

Migrating Your User Interface

The structure and implementation of the user interface in iOS is very different from that of Cocoa applications. Take, for example, the objects that represent views and windows in Cocoa. Although iOS and Cocoa both have objects representing views and windows, the way those objects work differs slightly on each platform. In addition, you must be more selective about what you display in your views because screen size is limited and views that handle touch events 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 iOS 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. When you want to present a new chunk of information, you push a new set of views onto the screen, replacing the previous set. This makes your interface design somewhat more complex, but because it is such a crucial way of displaying information, iOS provides considerable support for this type of organization.

View controllers in iOS are a critical part of managing your user interface. You use view controllers to structure your visual content, to present that content onto the screen, and to handle device-specific behaviors such as orientation changes. View controllers also manage views and work with the system to load and unload those views at appropriate times. 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 information about view controllers and how you use them to organize and manage your user interface, see *View Controller Programming Guide for iOS*. For general information about the user interface design principles of iOS, see *iOS 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 *View Programming Guide for iOS*.

Memory Management

In iOS, the preferred way to manage memory is automatic reference counting (ARC). With this model, the compiler does the memory management for you by automatically deallocating objects that are not being used by your code. All you have to do is maintain strong references to the objects you want to keep and set those references to `nil` when you no longer need the objects. You can also use the managed object model whereby you retain and release objects explicitly.

For more information on how to use ARC, see *Transitioning to ARC Release Notes*.

Framework Differences

Although most of the iOS frameworks are also present in 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 Cocoa developers might notice as they develop iOS applications.

UIKit Versus AppKit

In iOS, 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 iOS applications. Therefore, when migrating a Cocoa application to iOS, you must replace a significant number of interface-related classes and logic. Table 5-1 lists some of the specific differences between the frameworks to help you understand what is required of your application in iOS.

Table 5-1 Differences in interface technologies

Difference	Discussion
View classes	<p>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 iOS-based devices. Other views have more iOS-specific alternatives. For example, instead of the <code>NSBrowser</code> class, iOS uses an entirely different paradigm (navigation controllers) to manage the display of hierarchical information.</p> <p>For a description of the views and controls available in iOS, along with information on how to use them, see <i>iOS Human Interface Guidelines</i>.</p>

Difference	Discussion
View coordinate systems	<p>The drawing model for UIKit views is nearly identical to the model in AppKit, with one exception. AppKit views use 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 UIKit, the default origin point is in the top-left corner and the axes extend down and to the right. In AppKit, this coordinate system is known as a <i>modified coordinate system</i>, but for UIKit views it is the default coordinate system.</p> <p>For more information about view coordinate systems, see <i>View Programming Guide for iOS</i>.</p>
Windows as views	<p>Conceptually, windows and views represent the same constructs in UIKit as they do in AppKit. In implementation terms, however, the two platforms implement windows and views quite differently. In AppKit, the <code>NSWindow</code> class is a subclass of <code>NSResponder</code>, but in UIKit, the <code>UIWindow</code> class is actually a subclass of <code>UIView</code> instead. This change in inheritance means that windows in UIKit are backed by Core Animation layers and can perform most of the same tasks that views do.</p> <p>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.</p> <p>Another difference between iOS and OS X relates to the use of windows. Whereas a OS X application can have any number of windows, most iOS applications have only one. When you want to change the content displayed by your application, you swap out the views of your window rather than create a new window.</p>
Event handling	<p>The UIKit event-handling model is significantly different from the one found in AppKit. Instead of delivering mouse and keyboard events, UIKit delivers touch and motion 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.</p> <p>In iOS 3.2 and later, gesture recognizers provide a target-action model for responding to standard touch-based gestures such as taps, swipes, pinches, and rotations. You can also define your own gesture recognizers for custom gestures.</p> <p>For more information about handling events in iOS applications, see <i>Event Handling Guide for iOS</i>.</p>

Difference	Discussion
Target-action model	<p>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.</p> <p>For more information about the target-action model in iOS applications, see <i>Event Handling Guide for iOS</i>.</p>
Drawing and printing support	<p>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 iOS.</p> <p>In iOS 4.2 and later, applications can use the UIKit printing support to deliver data wirelessly to a nearby printer.</p> <p>For more information about graphics and drawing, see <i>Drawing and Printing Guide for iOS</i>.</p>
Text support	<p>The primary text support in iOS is geared toward composing email and notes. The UIKit classes let applications display and edit simple strings and somewhat more complex HTML content.</p> <p>In iOS 3.2 and later, more sophisticated text handling capabilities are provided through the Core Text and UIKit frameworks. You can use these frameworks to implement sophisticated text editing and presentation views and to support custom input methods for those views.</p> <p>In iOS 6, and later, UIKit includes support for styled text rendering using the <code>NSAttributedString</code> class. Several text-based views also allow you to specify attributed strings in addition to plain strings.</p> <p>For more information about text support, see <i>Text, Web, and Editing Programming Guide for iOS</i>.</p>
The use of accessor methods versus properties	<p>UIKit makes extensive use of properties throughout its class declarations. Properties were introduced to 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.</p> <p>For information about how to use properties, see “Declared Properties” in <i>The Objective-C Programming Language</i>.</p>

Difference	Discussion
Controls and cells	Controls in UIKit do not use cells. Cells are used in AppKit 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.
Table views	<p>The <code>UITableView</code> class in UIKit 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.</p> <p>In iOS 6, you can use the <code>UICollectionView</code> class to implement multicolumn and custom layouts.</p> <p>For more information about creating and using table views, see <i>Table View Programming Guide for iOS</i>.</p>
Menus	Nearly all applications written for iOS have a much smaller command set than do comparable OS X applications. For this reason, menu bars are not supported in iOS 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. For context-sensitive commands in iOS, you can display those on the edit menu in addition to (or in lieu of) commands such as Cut, Copy, and Paste.
Core Animation layers	<p>In iOS, 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 (and more directly) by changing a 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.</p> <p>For information about how Core Animation layers are integrated into the drawing model of iOS, see <i>View Programming Guide for iOS</i>.</p>

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

Foundation Framework Differences

A version of the Foundation framework is available in both OS X and iOS, 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. There are, however, some technologies that are not included in iOS. These technologies are listed in Table 5-2, along with the reasons the related classes are not available. Wherever possible, the table lists alternative technologies that you can use instead.

Table 5-2 Foundation technologies unavailable in iOS

Technology	Notes
Metadata and predicate management	In iOS 5 and later, the use of metadata queries is supported only for locating files in the user's iCloud storage. Prior to iOS 5, metadata queries are not supported at all.
Distributed objects and port name server management	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 <code>CFNetwork</code> frameworks to handle your networking needs.
Cocoa bindings	Cocoa bindings are not supported in iOS. Instead, iOS uses a slightly modified version of the target-action model that adds flexibility in how you handle actions in your code.
AppleScript support	AppleScript is not supported in iOS.

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 iOS. 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 OS X but not in iOS, see the class hierarchy diagram in “The Foundation Framework” in *Foundation Framework Reference*.

Changes to Other Frameworks

Table 5-3 lists the key differences in other frameworks found in iOS.

Table 5-3 Differences in frameworks common to iOS and OS X

Framework	Differences
AddressBook.framework	<p>This framework contains the interfaces for accessing user contacts. Although it shares the same name, the iOS version of this framework is very different from its OS X counterpart.</p> <p>In addition to the C-level interfaces for accessing contact data, in iOS, you can also use the classes of the Address Book UI framework to present standard picker and editing interfaces for contacts.</p> <p>For more information, see <i>Address Book Framework Reference for iOS</i>.</p>
AudioToolbox.framework AudioUnit.framework CoreAudio.framework	<p>The iOS 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 iOS, however, is the ability to trigger the vibrate option for iOS-based devices with the appropriate hardware.</p> <p>For information on how to use the audio support, see <i>Multimedia Programming Guide</i>.</p>
CFNetwork.framework	<p>This framework contains the Core Foundation Network interfaces. In iOS, the CFNetwork framework is a top-level framework and not a subframework. Most of the actual interfaces remain unchanged, however.</p> <p>For more information, see <i>CFNetwork Framework Reference</i>.</p>
CoreGraphics.framework	<p>This framework contains the Quartz interfaces. In iOS, 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 OS X. There are a few Quartz features that are not present in iOS, however, including PostScript support, image sources and destinations, Quartz Display Services support, and Quartz Event Services support.</p> <p>For more information, see <i>Core Graphics Framework Reference</i>.</p>

Framework	Differences
<code>OpenGL ES.framework</code>	<p>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 can perform most of the same operations you would normally perform 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 iOS.</p> <p>For information about the OpenGL ES support in iOS, see <i>OpenGL ES Programming Guide for iOS</i>.</p>
<code>QuartzCore.framework</code>	<p>This framework contains the Core Animation interfaces. Most of the Core Animation interfaces are the same for iOS and OS X. However, in iOS, 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 OS X version of the framework) are not available.</p> <p>For more information, see <i>Quartz Core Framework Reference</i>.</p>
<code>Security.framework</code>	<p>This framework contains the security interfaces. In iOS, it focuses on securing your application data by providing support for encryption and decryption, pseudorandom 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 OS X.</p> <p>For information about the security support, see <i>iOS App Programming Guide</i>.</p>
<code>SystemConfiguration.framework</code>	<p>This framework contains networking-related interfaces. In iOS, it 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.</p>

iOS Developer Tools

To develop applications for iOS, you need an Intel-based Macintosh computer and 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, though, and the following sections provide an introduction to the key applications you use to develop software for iOS.

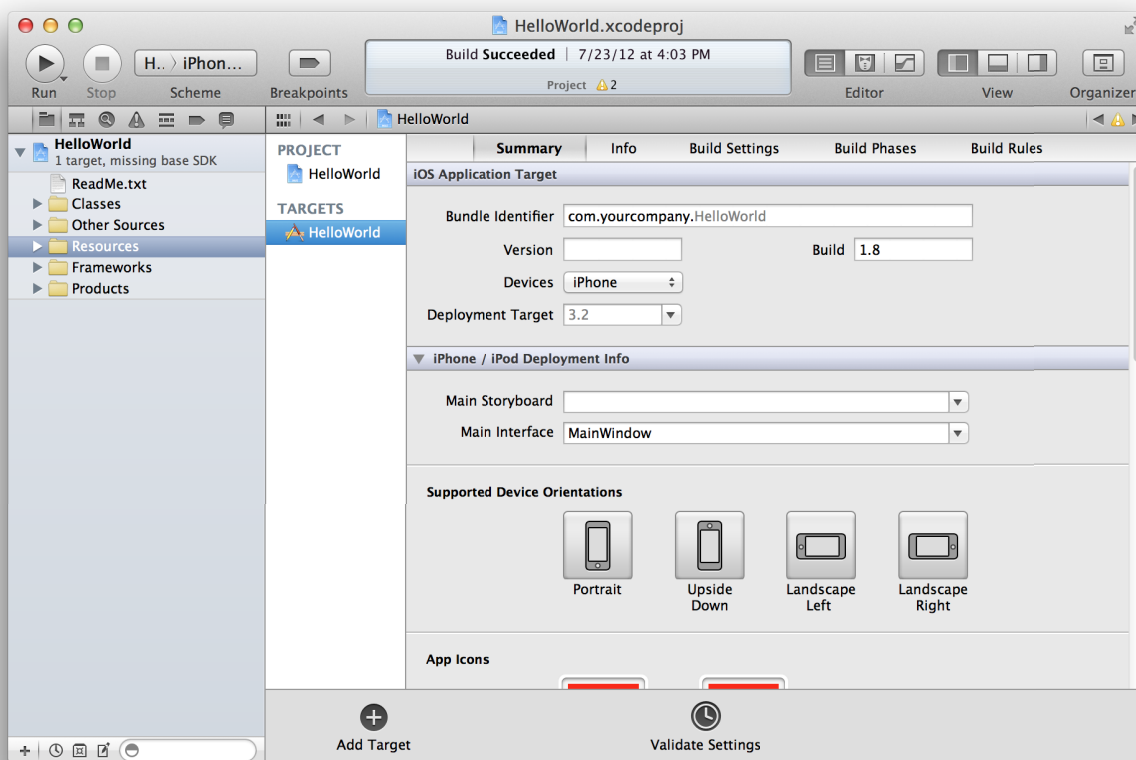
Xcode

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 iOS projects and source files, assemble your user interface, build your code into an executable, and run and debug your code either in iOS Simulator or on a device. Xcode incorporates a number of features to make developing iOS 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 integrated editor for creating storyboard and nib files
- 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
- LLVM and Clang support for C, C++, and Objective-C
- GCC compilers supporting C, C++, Objective-C, Objective-C++, and other languages
- A static analyzer for validating the behavior of your app and identifying potential problems.
- Integrated source-level debugging using GDB
- Support for integrated source-code management
- Support for DWARF and Stabs debugging information (DWARF debugging information is generated by default for all new projects)
- Support for managing iOS development devices.

To create a new iOS 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 & 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. You can then configure the workspace to display the panes you need for editing, navigating your project content, debugging, and obtaining additional information about items.

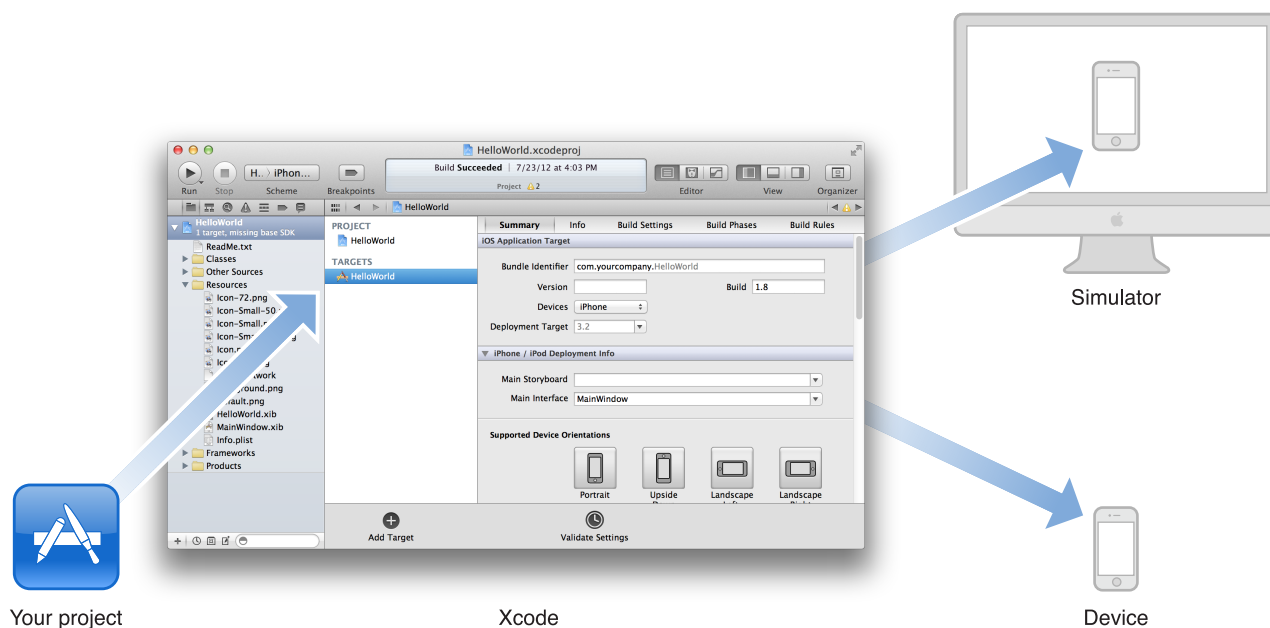
Figure A-1 An Xcode project window



When you build your application in Xcode, you have a choice of building it for iOS Simulator or for a device. 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 iOS-based device 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.

Figure A-2 Running a project from Xcode



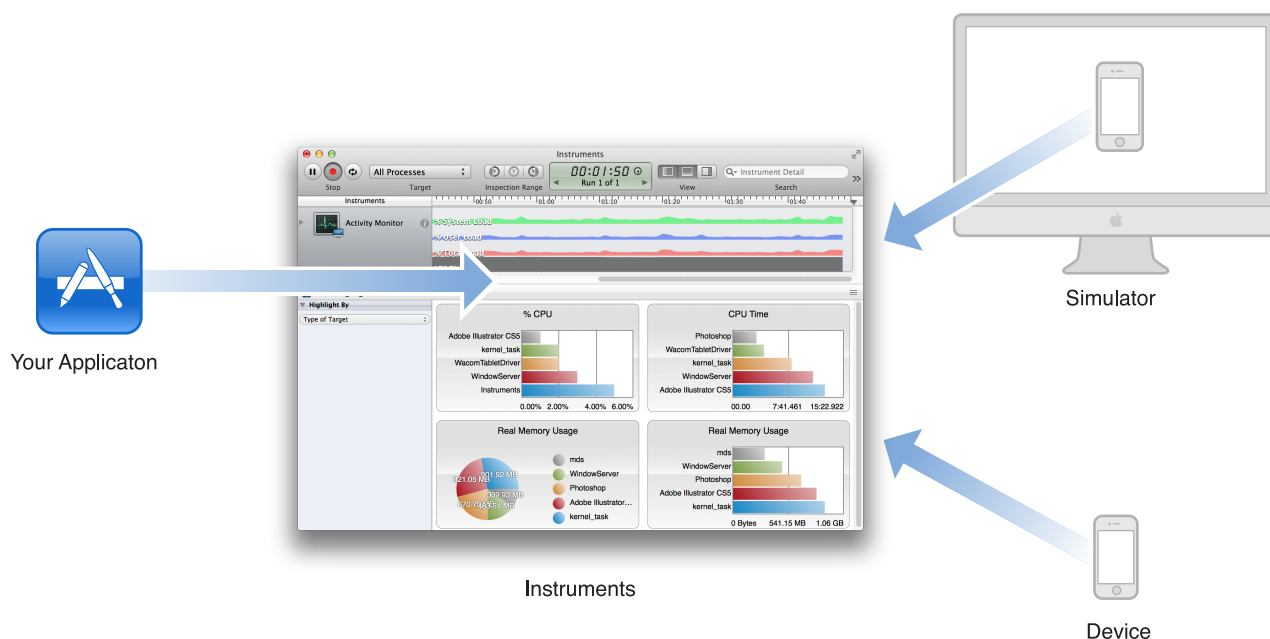
For details on how to build and run your project on iOS, see *Tools Workflow Guide for iOS*. For more information about the Xcode environment, see *Xcode 4 User Guide*.

Instruments

To ensure that you deliver the best user experience for your software, the Instruments environment lets you analyze the performance of your iOS applications while running in Simulator or on a device. Instruments gathers data from your running application and presents that data in a graphical display called the *timeline view*. You can gather data about your application's memory usage, disk activity, network activity, and graphics performance.

The timeline view can display all the 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.

Figure A-3 Using Instruments to tune your application



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 iOS applications, see *Tools Workflow Guide for iOS*. For general information on how to use Instruments, see *Instruments User Guide*.

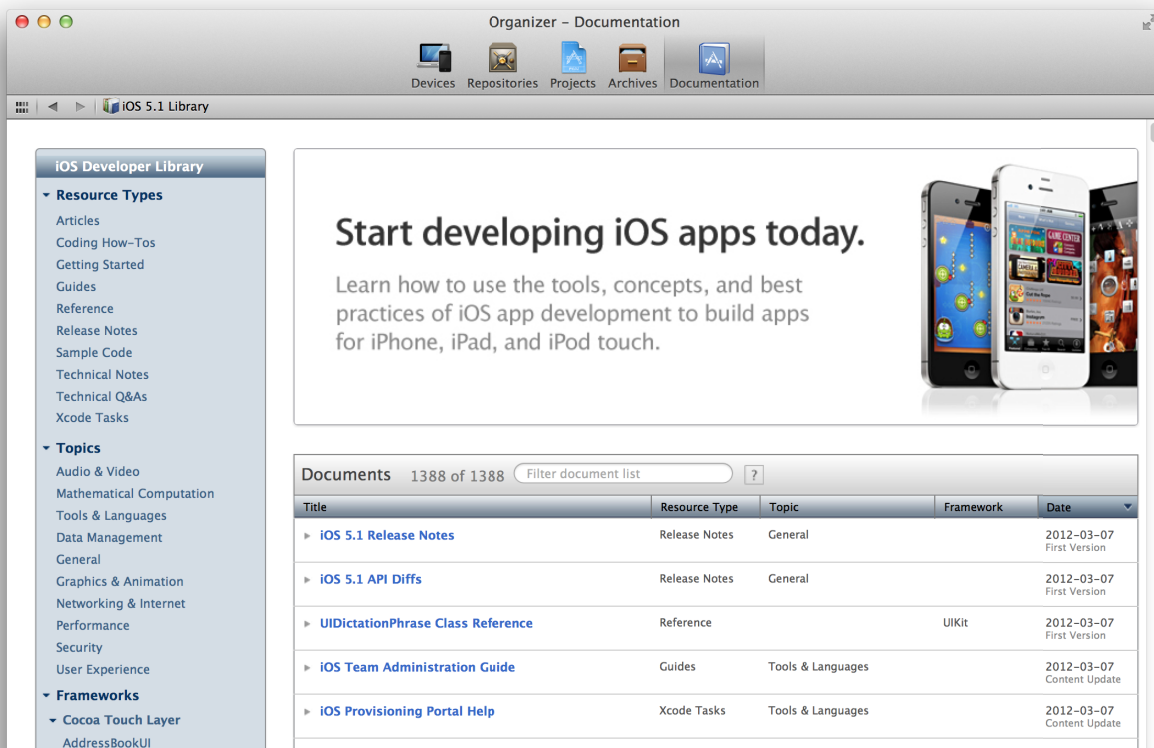
The Developer Library

The iOS Developer Library contains the documentation, sample code, tutorials, and other information you need to write iOS applications. Because the developer 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 developer library uses a few techniques for organizing content that should make it easier to browse.

You can access the iOS Developer Library from the [Apple Developer website](#) or from Xcode. In Xcode, choosing Help > Developer Documentation displays the Xcode documentation window, which is the central resource for accessing information about iOS development. You can use this window to browse the documentation, perform searches, and bookmark documents you may want to refer to later.

When you install the iOS SDK, Xcode automatically makes the iOS Developer Library available for you to use. (Xcode also downloads updates for you automatically, although you can change that setting in preferences.) The iOS Developer Library contains a lot of information so it is worth becoming at least somewhat familiar with its layout. Figure A-4 shows the main page of the developer library in the Xcode documentation window. The toolbar at the top of the page includes a search field and buttons for navigating around the documentation. You can browse the library by topic, by framework, or by the type of resource you are looking for. You can also use the filter field above the list of documents to narrow the set of displayed documents.

Figure A-4 The iOS Developer Library



Important: The content of the iOS Developer Library is updated regularly, but you can also access the latest documentation, release notes, Tech Notes, Technical Q&As, and sample code from the [iOS Dev Center](#). All documents are available in HTML and most are also available in PDF format.

Because the developer 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 a Quick Help pane in the Utilities section of the main project window. This pane shows you information about the designated symbol, including its syntax, description, and availability. It also shows you any related documentation and sample code resources. Clicking the links in this pane takes you to the corresponding resource in the developer library.

For more information about using the Documentation and Quick Help windows, see *Xcode 4 User Guide*.

iOS Frameworks

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

Device Frameworks

Table B-1 describes the frameworks available in iOS-based devices. You can find these frameworks in the `<Xcode.app> Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/<iOS_SDK> /System/Library/Frameworks` directory, where `<Xcode.app>` is the path to your Xcode app and `<iOS_SDK>` is the specific SDK version you are targeting. The "First available" column lists the iOS version in which the framework first appeared.

Table B-1 Device frameworks

Name	First available	Prefixes	Description
Accelerate.framework	4.0	cb1as, vDSP	Contains accelerated math and DSP functions. See "Accelerate Framework" (page 43).
Accounts.framework	5.0	AC	Contains interfaces for managing access to a user's system accounts. See "Accounts Framework" (page 35).
AddressBook.framework	2.0	AB	Contains functions for accessing the user's contacts database directly. See "Address Book Framework" (page 36).
AddressBookUI.framework	2.0	AB	Contains classes for displaying the system-defined people picker and editor interfaces. See "Address Book UI Framework" (page 16).
AdSupport.framework	6.0	AS	Contains a class for gathering analytics. See "Ad Support Framework" (page 36).

Name	First available	Prefixes	Description
<code>AssetsLibrary.framework</code>	4.0	AL	Contains classes for accessing the user's photos and videos. See "Assets Library Framework" (page 24).
<code>AudioToolbox.framework</code>	2.0	AU, Audio	Contains the interfaces for handling audio stream data and for playing and recording audio. See "Core Audio" (page 25).
<code>AudioUnit.framework</code>	2.0	AU, Audio	Contains the interfaces for loading and using audio units. See "Core Audio" (page 25).
<code>AVFoundation.framework</code>	2.2	AV	Contains Objective-C interfaces for playing and recording audio and video. See "AV Foundation Framework" (page 25).
<code>CFNetwork.framework</code>	2.0	CF	Contains interfaces for accessing the network via Wi-Fi and cellular radios. See "CFNetwork Framework" (page 36).
<code>CoreAudio.framework</code>	2.0	Audio	Provides the data types used throughout Core Audio. See "Core Audio" (page 25).
<code>CoreBluetooth.framework</code>	5.0	CB	Provides access to low-power Bluetooth hardware. See "Core Bluetooth Framework" (page 43)
<code>CoreData.framework</code>	3.0	NS	Contains interfaces for managing your application's data model. See "Core Data Framework" (page 37).
<code>CoreFoundation.framework</code>	2.0	CF	Provides fundamental software services, including abstractions for common data types, string utilities, collection utilities, resource management, and preferences. See "Core Foundation Framework" (page 37).
<code>CoreGraphics.framework</code>	2.0	CG	Contains the interfaces for Quartz 2D. See "Core Graphics Framework" (page 26).

Name	First available	Prefixes	Description
CoreImage.framework	5.0	CI	Contains interfaces for manipulating video and still images. See “Core Image Framework” (page 26).
CoreLocation.framework	2.0	CL	Contains the interfaces for determining the user’s location. See “Core Location Framework” (page 38).
CoreMedia.framework	4.0	CM	Contains low-level routines for manipulating audio and video. See “Core Media Framework” (page 38).
CoreMIDI.framework	4.2	MIDI	Contains low-level routines for handling MIDI data. See “Core Audio” (page 25).
CoreMotion.framework	4.0	CM	Contains interfaces for accessing accelerometer and gyro data. See “Core Motion Framework” (page 39).
CoreTelephony.framework	4.0	CT	Contains routines for accessing telephony-related information. See “Core Telephony Framework” (page 39).
CoreText.framework	3.2	CT	Contains a text layout and rendering engine. See “Core Text Framework” (page 27).
CoreVideo.framework	4.0	CV	Contains low-level routines for manipulating audio and video. Do not use this framework directly.
EventKit.framework	4.0	EK	Contains interfaces for accessing a user’s calendar event data. See “Event Kit Framework” (page 39).
EventKitUI.framework	4.0	EK	Contains classes for displaying the standard system calendar interfaces. See “Event Kit UI Framework” (page 16).
External-Accessory.framework	3.0	EA	Contains interfaces for communicating with attached hardware accessories. See “External Accessory Framework” (page 44).

Name	First available	Prefixes	Description
Foundation.framework	2.0	NS	Contains interfaces for managing strings, collections, and other low-level data types. See “Foundation Framework” (page 39).
GameKit.framework	3.0	GK	Contains the interfaces for managing peer-to-peer connectivity. See “Game Kit Framework” (page 17).
GLKit.framework	5.0	GLK	Contains Objective-C utility classes for building complex OpenGL ES applications. See “GLKit Framework” (page 28).
GSS.framework	5.0	gss	Provides a standard set of security-related services.
iAd.framework	4.0	AD	Contains classes for displaying advertisements in your application. See “iAd Framework” (page 17).
ImageIO.framework	4.0	CG	Contains classes for reading and writing image data. See “Image I/O Framework” (page 27).
IOKit.framework	2.0	N/A	Contains interfaces used by the device. Do not include this framework directly.
MapKit.framework	3.0	MK	Contains classes for embedding a map interface into your application and for reverse-geocoding coordinates. See “Map Kit Framework” (page 18).
MediaPlayer.framework	2.0	MP	Contains interfaces for playing full-screen video. See “Media Player Framework” (page 28).
MediaToolbox.framework	6.0	MT	Contains interfaces for playing audio content.
MessageUI.framework	3.0	MF	Contains interfaces for composing and queuing email messages. See “Message UI Framework” (page 18).

Name	First available	Prefixes	Description
MobileCoreServices.framework	3.0	UT	Defines the uniform type identifiers (UTIs) supported by the system. See “Mobile Core Services Framework” (page 40).
NewsstandKit.framework	5.0	NK	Provides interfaces for downloading magazine and newspaper content in the background. See “Newsstand Kit Framework” (page 40).
OpenAL.framework	2.0	AL	Contains the interfaces for OpenAL, a cross-platform positional audio library. See “OpenAL Framework” (page 29).
OpenGLES.framework	2.0	EAGL, GL	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” (page 29).
PassKit.framework	6.0	PK	Contains interfaces for creating digital passes to replace things like tickets, boarding passes, member cards, and more. See “Pass Kit Framework” (page 41).
QuartzCore.framework	2.0	CA	Contains the Core Animation interfaces. See “Quartz Core Framework” (page 29).
QuickLook.framework	4.0	QL	Contains interfaces for previewing files. See “Quick Look Framework” (page 41).
Security.framework	2.0	CSSM, Sec	Contains interfaces for managing certificates, public and private keys, and trust policies. See “Security Framework” (page 44).
Social.framework	6.0	SL	Contains interfaces for interacting with social media accounts. See “Social Framework” (page 41).

Name	First available	Prefixes	Description
StoreKit.framework	3.0	SK	Contains interfaces for handling the financial transactions associated with in-app purchases. See “Store Kit Framework” (page 42).
System-Configuration.framework	2.0	SC	Contains interfaces for determining the network configuration of a device. See “System Configuration Framework” (page 42).
Twitter.framework	5.0	TW	Contains interfaces for sending tweets via the Twitter service. See “Twitter Framework” (page 18).
UIKit.framework	2.0	UI	Contains classes and methods for the iOS application user interface layer. See “UIKit Framework” (page 19).
VideoToolbox.framework	6.0	N/A	Contains interfaces used by the device. Do not include this framework directly.

Simulator Frameworks

Although you should always target the device frameworks when writing your code, you might need to compile your code specially for Simulator during testing. The frameworks available on the device and in Simulator are mostly identical, but there are a handful of differences. For example, Simulator uses several 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 because of system limitations. For a list of frameworks and for information about the specific differences between the device and Simulator frameworks, see *Tools Workflow Guide for iOS*.

System Libraries

Note that some specialty libraries at the Core OS and Core Services level are not packaged as frameworks. Instead, iOS 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 iOS 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 iOS, look in

`<Xcode.app> /Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/<iOS_SDK> /usr/lib`, where `<Xcode.app>` is the path to your Xcode app and `<iOS_SDK>` is the specific SDK version you are targeting.

For example, the shared libraries for the iOS 6.0 SDK would be located in the

`/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS6.0.sdk/usr/lib` directory, with the corresponding headers in

`/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS6.0.sdk/usr/include`

iOS uses symbolic links to point to the 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 iOS; if your software is linked to a specific version, that version might not always be available on the user's system.

Document Revision History

This table describes the changes to *iOS Technology Overview*.

Date	Notes
iOS 6.0	Contains information about new frameworks and technologies introduced in iOS 6.
2011-10-12	Added technologies introduced in iOS 5.
2010-11-15	Updated the document to reflect new features in iOS 4.1 and iOS 4.2.
2010-07-08	Changed the title from "iPhone OS Technology Overview."
2010-06-04	Updated to reflect features available in iOS 4.0.
2009-10-19	Added links to reference documentation in framework appendix.
2009-05-27	Updated for iOS 3.0.
2008-10-15	New document that introduces iOS and its technologies.



Apple Inc.

© 2012 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.

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

Apple, the Apple logo, AirPlay, Apple TV, AppleScript, Bonjour, Cocoa, Cocoa Touch, Instruments, iPad, iPhone, iPod, iPod touch, iTunes, Keychain, Mac, Macintosh, Objective-C, OS X, Pages, Passbook, Quartz, Safari, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

Retina is a trademark of Apple Inc.

iAd, iCloud, and iTunes Store are service marks of Apple Inc., registered in the U.S. and other countries.

App Store is a service mark of Apple Inc.

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

Java is a registered trademark of Oracle and/or its affiliates.

OpenGL is a registered trademark of Silicon Graphics, Inc.

UNIX is a registered trademark of The Open Group.

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

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.