iOS Technology Overview

General

2010-07-08

Contents

Figures and Tables

Introduction

iOS is the operating system at the heart of iPhone, iPod touch, and iPad devices.

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

Who Should Read 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 system features or about features that have no impact on the software development process.

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

Organization of This Document

This document has the following chapters and appendixes:

- "About iOS Development" (page 13) provides a high level overview of iOS and developing applications for it using the iPhone SDK.
- "Cocoa Touch Layer" (page 19) provides a look at the Cocoa Touch layer in iOS and the features it provides to your applications.
- "Media Layer" (page 27) provides a look at the Media layer in iOS and the features it provides to your applications.
- "Core Services Layer" (page 35) provides a look at the Core Services layer in iOS and the features it provides to your applications.
- "Core OS Layer" (page 43) provides a look at the Core OS layer in iOS and the features it provides to your applications.
- "Migrating from Cocoa" (page 45) provides starter advice for developers who are migrating an existing Cocoa application to iOS.
- "iOS Frameworks" (page 59) describes the frameworks you can use to develop your software. Use this information to find specific technologies or to find when a given framework was introduced to iOS.
- "iOS Developer Tools" (page 53) provides an overview of the available applications you can use to create software for iOS.

Getting the iPhone SDK

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

For additional information about the tools available for working with Mac OS X and its technologies, see "iOS Developer Tools" (page 53).

Providing Feedback

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

If you encounter bugs in Apple software or documentation, you are encouraged to report them to Apple. You can also file enhancement requests to indicate features you would like to see in future revisions of a product or document. To file bugs or enhancement requests, go to the Bug Reporting page of the Apple Developer website:

http://developer.apple.com/bugreporter/

To file bugs, you must be registered as an Apple Developer. You can obtain a login name for free by following the instructions on the Apple Developer Registration page.

See Also

The following documents provide key information related to iOS development:

- *Cocoa Fundamentals Guide* provides fundamental information about the design patterns and practices used to develop iOS applications.
- *iOS Application Programming Guide* provides an architectural overview of iOS applications along with practical guidance on how to create them.
- *iPhone Human Interface Guidelines* and *iPad Human Interface Guidelines* provide guidance and important information about how to design your application's user interface.
- *iOS Development Guide* provides important information about the iOS development process from the tools perspective. This document covers the configuration of devices and the use of Xcode (and other tools) for building, running, and testing your software.
- *The Objective-C Programming Language* introduces Objective-C and the Objective-C runtime system, which is the basis of much of the dynamic behavior and extensibility of iOS.

About iOS Development

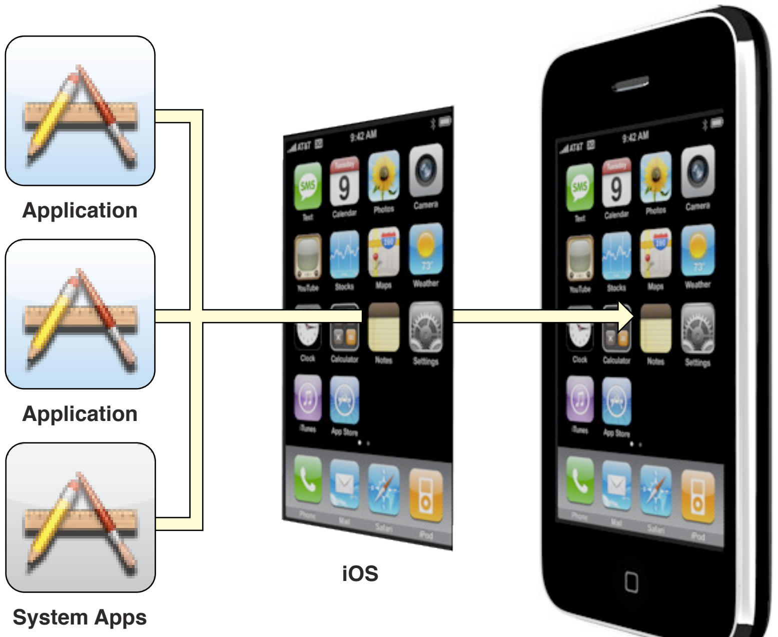
iOS is the operating system that runs on iPhone, iPod touch, and iPad devices. This operating system manages the device hardware and also provides the basic technologies needed to implement native applications on the phone. Depending on the device, the operating system also ships with several system applications, such as Phone, Mail, and Safari, that provide standard system services for the user.

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

The iOS Architecture

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

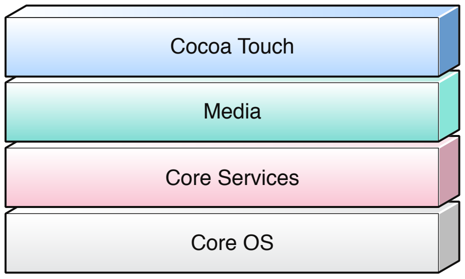
Figure 1-1 Applications layered on top of iOS



Note: Even though your application is generally protected from changes to the underlying hardware, you still need to account for some differences between devices in your code. For example, an iPad or iPod touch may not be able to open URLs containing a phone number while an iPhone can.

The implementation of iOS technologies can be viewed as a set of layers, which are shown in "Game Kit Framework." At the lower layers of the system are the fundamental services on which all applications rely, while higher-level layers contain more sophisticated services and technologies.

Figure 1-2 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 number of lines of code you have to write and encapsulate potentially complex features, such as sockets and threads. Although they abstract out lower-level technologies, they do not mask those technologies from you. The lower-level frameworks are still available for developers who prefer using them or who want to use aspects of those frameworks that are not exposed at the higher level.

The technologies and frameworks for each layer are described in later chapters of this document.

What's in the iPhone SDK?

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

Apple delivers most of its system interfaces in special packages called frameworks. A **framework** is a directory that contains a dynamic shared library and the resources (such as header files, images, helper applications, and so on) needed to support that library. To use frameworks, you link them into your application project just as you would any other shared library. Linking them to your project gives you access to the features of the framework and also lets the development tools know where to find the header files and other framework resources.

In addition to frameworks, Apple also delivers some technologies in the form of standard shared libraries. Because iOS is based on UNIX, many of the technologies that form the lower-levels of the operating system are derived from open-source technologies. The interfaces for these technologies are therefore available in the standard library and interface directories.

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

- Xcode Tools - provides the tools that support iOS application development, including the following key applications:
  - **Xcode** - an integrated development environment that manages your application projects and lets you edit, compile, run, and debug your code. Xcode integrates with many other tools and is the main application you use during development.
  - **Interface Builder** - a tool you use to assemble your user interface visually. The interface objects you create are then saved to a special resource file format and loaded into your application at runtime.
  - **Instruments** - a runtime performance analysis and debugging tool. You can use Instruments to gather information about your application's runtime behavior and identify potential problems.
- **iPhone Simulator** - a Mac OS X application that simulates the iOS technology stack, allowing you to test iOS applications locally on your Intel–based Macintosh computer.
- **iOS Reference Library**- the SDK includes the reference documentation for iOS by default. Updates to this library are also downloaded automatically when they become available. To display the reference library, choose Help > Developer Documentation.

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

For information on how to install the iPhone SDK and use it for developing iOS applications, see *iOS Development Guide*. For more information about the frameworks in iOS, and for information about where to find the low-level system libraries, see "iOS Frameworks" (page 59).

What Can You Create?

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

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

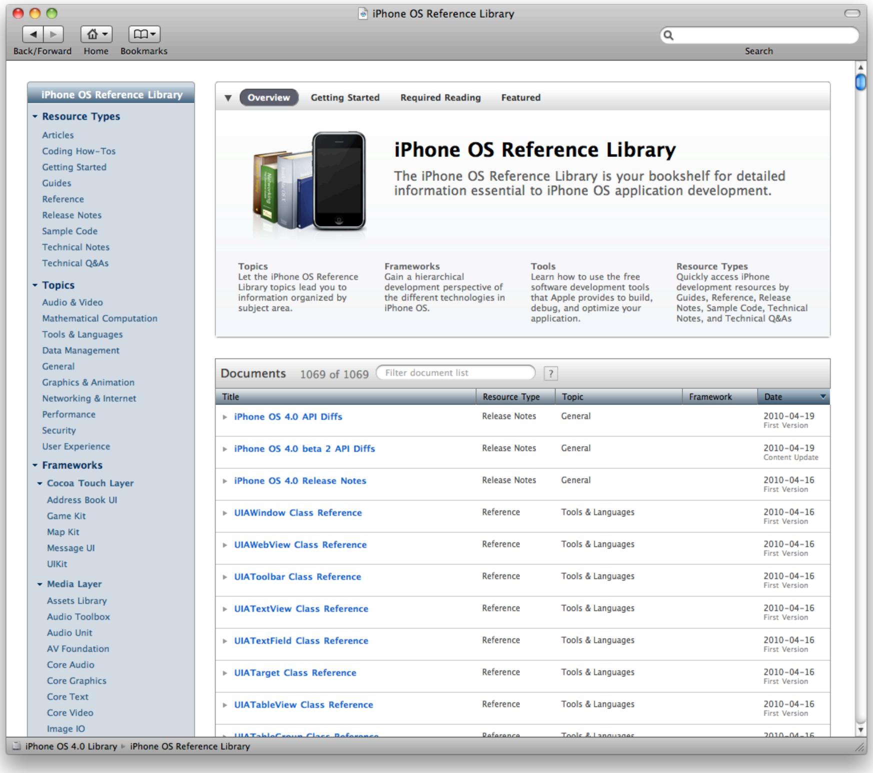
How to Use the Reference Library

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

You can access the iOS Reference 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. Documents are grouped by content into doc sets to facilitate updates and to scope searches to only the relevant set of documents.

When you install the iPhone SDK, Xcode automatically installs the docset containing the iOS Reference Library for you to use. (Xcode also downloads docset updates for you automatically, although you can change that setting in preferences.) The iOS Reference Library contains a lot of information so it is worth becoming at least somewhat familiar with its layout. Figure 1-3 shows the main page of the reference library in the Xcode documentation window. The toolbar at the top of the page include a search field and buttons for navigating to other installed docsets and to any bookmarks you created. 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 control above the list of documents to narrow the set of displayed documents.

Figure 1-3 The iOS Reference Library



iPhone OS Reference Library

The iPhone OS Reference Library is your bookshelf for detailed information essential to iPhone OS application development.

Topics
Let the iPhone OS Reference Library topics lead you to information organized by subject area.

Frameworks
Gain a hierarchical development perspective of the different technologies in iPhone OS.

Tools
Learn how to use the free software development tools that Apple provides to build, debug, and optimize your application.

Resource Types
Quickly access iPhone development resources by Guides, Reference, Release Notes, Sample Code, Technical Notes, and Technical Q&As

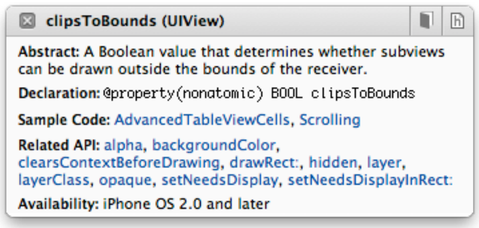
Documents 1069 of 1069

Title	Resource Type	Topic	Framework	Date
iPhone OS 4.0 API Diffs	Release Notes	General		2010-04-19 First Version
iPhone OS 4.0 beta 2 API Diffs	Release Notes	General		2010-04-19 Content Update
iPhone OS 4.0 Release Notes	Release Notes	General		2010-04-16 First Version
UIAWindow Class Reference	Reference	Tools & Languages		2010-04-16 First Version
UIAWebView Class Reference	Reference	Tools & Languages		2010-04-16 First Version
UIAToolbar Class Reference	Reference	Tools & Languages		2010-04-16 First Version
UIATextView Class Reference	Reference	Tools & Languages		2010-04-16 First Version
UIATextField Class Reference	Reference	Tools & Languages		2010-04-16 First Version
UIATarget Class Reference	Reference	Tools & Languages		2010-04-16 First Version
UIATableView Class Reference	Reference	Tools & Languages		2010-04-16 First Version
UIATableGroup Class Reference	Reference	Tools & Languages		2010-04-16

Important: The content of the iOS Reference Library is updated regularly, but you can also access the latest documentation, release notes, Tech Notes, Technical Q&As, and sample code from the iPhone Dev Center (http://developer.apple.com/iphone). All documents are available in HTML and most are also available in PDF format.

Because the reference library provides a tremendous amount of information, sorting through all that information while you are trying to write code can be cumbersome. To help you find specific information quickly, Xcode also provides a Quick Help window, shown in Figure 1-4. This window 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 window takes you to the corresponding resource in the reference library. To display this window, hold down the Option key and double-click a symbol in the Xcode editor window.

Figure 1-4 Quick help in Xcode



clipsToBounds (UIView)

Abstract: A Boolean value that determines whether subviews can be drawn outside the bounds of the receiver.

Declaration: `@property(nonatomic) BOOL clipsToBounds`

Sample Code: AdvancedTableViewCells, Scrolling

Related API: alpha, backgroundColor, clearsContextBeforeDrawing, drawRect:, hidden, layer, layerClass, opaque, setNeedsDisplay, setNeedsDisplayInRect:

Availability: iPhone OS 2.0 and later

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

Cocoa Touch Layer

The **Cocoa Touch** layer contains the key frameworks for building iOS applications. The technologies in this layer provide the infrastructure you need to implement the visual interface of your application and interact with many high-level system services. When developing your applications, you should always start with these frameworks and drop down to lower-level frameworks only as needed.

High-Level Features

The following sections describe some of the more common features you might want to support in your applications.

Multitasking

Applications built using iPhone SDK 4.0 or later (and running in iOS 4.0 and later) are no longer terminated when the user presses the Home button; instead, they now shift to a background execution context. For many applications, this means that the application enters a suspended state of execution shortly after entering the background. Keeping the application in memory avoids the subsequent launch cycle and allows an application to simply reactivate itself, which improves the overall user experience. And suspending the application improves overall system performance by minimizing power usage and giving more execution time to the foreground application.

Although most applications are suspended shortly after moving to the background, applications that need to continue working in the background may do so using one of the following techniques:

- 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 some additional work on your part. Background applications can still be terminated under certain conditions (such as during low-memory conditions), and so applications must be ready to exit at any time. This means that many of the tasks you used to perform at quit time must now be performed when your application moves to the background. This requires implementing some new methods in your application delegate to respond to application state transitions.

For more information on how to handle the new background state transitions, and for information on how to continue running in the background, see *iOS Application Programming Guide*.

Data Protection

Applications that work with sensitive user data can now take advantage of the built-in encryption available on some devices to protect that data. When your application designates a particular 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.

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

For more information about how to add data protection to the files of your application, see "Implementing Standard Application Behaviors" in *iOS Application Programming Guide*.

Apple Push Notification Service

In iOS 3.0 and later, the Apple Push Notification Service provides a way to alert your users of new information, even when your application is not actively running. Using this service, you can push text notifications, trigger audible alerts, or add a numbered badge to your application icon. These messages let users know that they should open your application to receive the related information.

From a design standpoint, there are two parts to making push notifications work for your iOS applications. First, you need to request the delivery of notifications to your iOS application and then you need to configure your application delegate to process them. The delegate works together with the shared `UIApplication` object to perform both of these tasks. Second, you need to provide a server-side process to generate the notifications in the first place. This process lives on your own local server and works with Apple Push Notification Service to trigger the notifications.

For more information about how to configure your application to use remote notifications, see *Local and Push Notification Programming Guide*.

Local Notifications

Introduced in iOS 4.0, local notifications complement the existing push notifications by giving applications an avenue for generating the notifications locally instead of relying on an external server. Background applications 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. After attaching it to your view, you tell it what action you want performed when the gesture occurs. The gesture recognizer object then tracks the raw events and applies the system-defined heuristics for what the given gesture should be. Prior to gesture recognizers, the process for detecting a gesture involved tracking the raw stream of touch events coming to your view and applying potentially complicated heuristics to determine whether the events represented the given gesture.

UIKit now 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 system-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*.

File-Sharing Support

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

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

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

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

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

Peer to Peer Services

In iOS 3.0 and later, peer-to-peer connectivity over Bluetooth is provided by the Game Kit framework. 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 these features in other types of applications.

For information about how to use peer-to-peer connectivity features in your application, see *Game Kit Programming Guide*. For an overview of the Game Kit framework, see "Game Kit Framework" (page 23).

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 so as 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 the corresponding framework reference.

External Display Support

Introduced in iOS 3.2, iOS–based devices can 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.

- The `UIScreen` class provides support for retrieving screen objects for all available screens (including the device's main screen). Each screen object contains information about the properties of the screen itself, including the dimensions that correctly take into account the size and pixel aspect ratio of the screen.
- The `UIScreenMode` class provides information about one particular size and pixel aspect ratio setting of a screen.
- Windows (represented by the `UIWindow` class) can now be assigned to a specific screen. To mirror content, you must provide two separate windows and display the same content in both.

For more information about the support offered by these classes, see the individual class descriptions in *UIKit Framework Reference*.

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 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, 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 non-game applications as well. The framework provides you with networking features through a simple (yet powerful) set of classes built on top of Bonjour. These classes abstract out many of the network details, making it easy for developers who might be inexperienced with networking programming to incorporate networking features into their applications.

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

iAd Framework

Introduced in iOS 4.0, the iAd (`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 Framework Reference*.

Map Kit Framework

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

In iOS 4.0, this framework added 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. 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.

For more information about the classes of the Map Kit framework, see *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 can present in your application. You can populate the fields of this interface with the contents of the message you want to send. You can set the recipients, subject, body content, and any attachments you want to include with the message. The user then has the option of editing the message prior to accepting it. Once accepted, the message is queued in the user's outbox for delivery.

In iOS 4.0 and later, this framework provides a view controller for presenting an SMS composition panel. 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*.

UIKit Framework

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

- Application management
- User interface management
- Graphics and windowing support
- Multitasking support
- 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
- Support for the Apple push notification service; see "Apple Push Notification Service" (page 20)
- Accessibility support for disabled users
- Local notification scheduling and delivery
- 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

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

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

Graphics Technologies

High-quality graphics are an important part of all 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 what is offered by UIKit and provide custom behaviors. 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.
- Core Animation (part of the Quartz Core framework) provides advanced support for animating views and other content.
- OpenGL ES provides support for 2D and 3D rendering using hardware-accelerated interfaces.
- Core Text provides a sophisticated text layout and rendering engine.
- Image I/O provides interfaces for reading and writing most image formats.
- The Assets Library framework provides access to the photos and videos in the user's photo library.

For the most part, applications running on devices with high-resolution screens should work with little or no modifications. The coordinate values you specify during drawing or when manipulating views are all mapped to a logical coordinate system, which is decoupled from the underlying screen resolution. 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 "Supporting High-Resolution Screens" in *iOS Application Programming Guide*.

For information about the graphics-related frameworks, see the corresponding entries in "Media Layer Frameworks" (page 29).

Audio Technologies

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

The system provides several ways to play back and record audio content depending on your needs. When choosing an audio technology, remember that the higher-level frameworks simplify the work you have to do to support audio playback 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 Media Player framework provides easy access to the user's iTunes library and support for playing tracks and playlists.
- AV Foundation provides a set of easy-to-use Objective-C interfaces for managing audio playback and recording.
- OpenAL provides a set of cross-platform interfaces for delivering positional audio.
- 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.

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 29).

Video Technologies

Whether you are playing movie files from your application bundle or streamed content from the network, iOS provides several technologies to play those movies. 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 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 Media Player framework provides a set of simple-to-use interfaces for presenting full- or partial-screen movies from your application.
- AV Foundation provides a set of Objective-C interfaces for managing the capture and playback of movies.
- Core Media describes the low-level types used by the higher-level frameworks and provides low-level interfaces for manipulating media.

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, 48kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats
- H.264 video, up to 768 Kbps, 320 by 240 pixels, 30 frames per second, Baseline Profile up to Level 1.3 with AAC-LC audio up to 160 Kbps, 48kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats
- MPEG-4 video, up to 2.5 Mbps, 640 by 480 pixels, 30 frames per second, Simple Profile with AAC-LC audio up to 160 Kbps, 48kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats
- Numerous audio formats, including the ones listed in "Audio Technologies" (page 28)

For information about each of the audio frameworks, see the corresponding entry in "Media Layer Frameworks" (page 29).

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 a user's photos and videos. Using this framework, you can access the same assets that are nominally 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 this support to play file- or memory-based sounds of any duration. You can play multiple sounds simultaneously and control various playback aspects of each sound. In 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 significantly 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

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.

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 3-1. **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 access the vibrate capability on devices that support it.

Table 3-1 Core Audio frameworks

Framework	Services
`CoreAudio.framework`	Defines the audio data types used throughout Core Audio.
`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.
`AudioUnit.framework`	Provides services for using the built-in audio units, which are audio processing modules.

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

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 Mac OS X. It provides support for path-based drawing, anti-aliased rendering, gradients, images, colors, coordinate-space transformations, and PDF document creation, display, and parsing. Although the API is C based, it uses object-based abstractions to represent fundamental drawing objects, making it easy to store and reuse your graphics content.

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

Core 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 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 Core Media. Most applications should 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 is built on top of the Core Graphics data types and functions and supports all of the standard image types available in iOS.

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

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 playback video using a standard system interface. In iOS 3.0, support was added for accessing the user's iTunes 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 this framework 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 configuring and managing movie playback.

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

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

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

OpenGL ES Framework

The **OpenGL ES framework** (`OpenGLES.framework`) provides tools for drawing 2D and 3D content. It is a C-based framework that works closely with the device hardware to provide high frame rates for full-screen game-style applications.

You always use the OpenGL framework in conjunction with the EAGL interfaces. These interfaces are part of the OpenGL ES framework and provide the interface between your OpenGL ES drawing code and the native window objects of your application.

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

High-Level Features

The following sections describe some of the more common features you might want to support in your applications.

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 how you use them, see *A Short Practical Guide to Blocks*. For more information about blocks, see *Blocks Programming Topics*.

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, monitoring signals and process events, and more.

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.

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 40).

Location Services

Applications that want to track the user's position can do so using the interfaces of the Core Location framework. This framework takes advantage of all the available hardware radios (including Wi-Fi, cellular, and GPS where available) to report the user's current location. Applications can use this information to tailor the information they deliver to the user or to implement specific features. For example, a social application might allow you to find other nearby users of the application and communicate with them.

For more information about using location services, see *Location Awareness Programming Guide*. For more information about the Core Location framework, see also "Core Location Framework" (page 38).

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` libraries. This open source library lets you parse or write arbitrary XML data quickly and transform XML content to HTML.

The header files for accessing the `libXML2` library are located in the <*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.

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.

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

CFNetwork Framework

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

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

CFNetwork is based, both physically and theoretically, on BSD sockets. For information on how to use CFNetwork, see *CFNetwork Programming Guide* 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 where the data model is already highly structured. Instead of defining data structures programmatically, you use the graphical tools in Xcode to build a schema representing your data model. At runtime, instances of your data-model entities are created, managed, and made available through the Core Data framework.

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

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

If you are starting to develop a new application or are planning a significant update to an existing application, you should consider using Core Data. For an example of how to use Core Data in an 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 bridged with.

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

Core Location Framework

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

In iOS 3.0, support was added for accessing compass information on iOS–based devices that include suitable hardware.

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 the classes of the Core Location framework, see *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 AV Foundation. Most applications should 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 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 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.

For more information about the classes and methods of this framework, see *Event Kit Framework Reference*. See also "Event Kit UI Framework" (page 23).

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 38). 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*.

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*.

Store Kit Framework

Introduced in iOS 3.0, 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 if 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.

Core OS Layer

The Core OS layer provides 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 math, big-number, and DSP calculations, among others. The advantage of using this framework over writing your own versions of these libraries is that it is optimized for the different 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*.

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 the classes of the External Accessory framework, see *External Accessory Framework Reference*. For information about developing accessories for iOS-based devices, go to http://developer.apple.com.

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 pseudo random numbers. It also supports the storage of certificates and cryptographic keys in the keychain, which is a secure repository for sensitive user data.

The CommonCrypto interfaces provide additional support for symmetric encryption, HMAC, and Digests. The Digests feature provides functions that are essentially compatible with the functionality normally found in the OpenSSL library, which is not available in 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 *iOS Development Guide*. For information about the entitlements you can configure, see the description for the `SecItemAdd` function in *Keychain Services Reference*.

System

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

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 an existing Cocoa developer, many of the frameworks available in iOS should seem familiar to you. The basic technology stack in iOS is identical in many respects to the one found in Mac OS X. Despite the similarities, however, the frameworks in iOS are not exactly the same as their Mac 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. For information about designing applications for iOS, see *iOS Application Programming Guide*.

Migrating Your Data Model

Cocoa applications whose data model is based on classes in the Foundation and Core Foundation frameworks can be brought over to iOS with little or no modification. Both frameworks are supported in iOS and are virtually identical to their Mac OS X counterparts, although there are some differences. However, most of the differences are relatively minor or are related to features that would need to be removed in the 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 49).

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 limited screen space, and may also slow down your application, because of the extra work required to load that data. Refactoring your Cocoa application's data structures might be worthwhile if it provides better performance and a better user experience in iOS.

Migrating Your User Interface

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

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

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

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

Memory Management

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

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

Framework Differences

Although most of the iOS frameworks are also present in Mac OS X, there are platform differences in how those frameworks are implemented and used. The following sections call out some of the key differences that existing Mac OS X developers might notice as they develop 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. For this reason, when migrating a Cocoa application to iOS, you must replace a significant number of interface-related classes and logic. Table 6-1 lists some of the specific differences between the frameworks to help you understand what is required of your application in iOS.

Table 6-1 Differences in interface technologies

Difference	Discussion
Document support	In iOS, the role of documents is deemphasized in favor of a simpler content model. Because applications typically have only one window (unless an external display is connected), the main window acts as the sole environment for creating and editing all application content. More importantly, the creation and management of any actual document-related files is handled behind the scenes by the application and not exposed to the user.
View classes	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 `NSBrowser` class, iOS uses an entirely different paradigm (navigation controllers) to manage the display of hierarchical information. For a description of the views and controls available in iOS along with information on how to use them, see *iPhone Human Interface Guidelines*.
View coordinate systems	In iOS, the drawing model for Quartz and UIKit content is nearly identical to the model in Mac OS X, with one exception. The Mac OS X drawing model uses a coordinate system where the origin for windows and views is in the lower-left corner by default, with axes extending up and to the right. In iOS, the default origin point is in the top-left corner and the axes extend down and to the right. In Mac OS X, this coordinate system is known as a “flipped” coordinate system, but in iOS it is the default coordinate system. For more information about graphics and coordinate systems, see *View Programming Guide for iOS*
Windows as views	Conceptually, windows and views represent the same constructs in iOS as they do in Mac OS X. In implementation terms, however, the two platforms implement windows and views quite differently. In Mac OS X, the `NSWindow` class is a subclass of `NSResponder`, but in iOS, the `UIWindow` class is actually a subclass of `UIView` instead. This change in inheritance means that windows use Core Animation layers to implement their drawing surface. The main reason for having window objects at all in UIKit is to support the layering of windows within the operating system. For example, the system displays the status bar in a separate window that floats above your application’s window. Another difference between iOS and Mac OS X relates to the use of windows. Whereas a Mac OS X application can have any number of windows, most iOS applications have only one. Displaying different screens of information in an iOS application is done by swapping out custom views from the application window rather than by changing the window.

Difference	Discussion
Event handling	The UIKit event-handling model is significantly different from the one found in Mac OS X. Instead of mouse and keyboard events, UIKit delivers touch 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. For more information about handling events in iOS applications, see *Event Handling Guide for iOS*.
Target-action model	UIKit supports three variant forms for action methods, as opposed to just one for AppKit. Controls in UIKit can invoke actions for different phases of the interaction and they have more than one target assigned to the same interaction. Thus, in UIKit a control can deliver multiple distinct actions to multiple targets over the course of a single interaction cycle. For more information about the target-action model in iOS applications, see *Event Handling Guide for iOS*.
Drawing and printing support	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. Printing is not supported because there is no support for connecting to printers or other print-related hardware from an iOS-based device. For more information about graphics and drawing, see *View Programming Guide for iOS*
Text support	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. 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. For more information about text support, see *Text and Web Programming Guide for iOS*.
The use of accessor methods versus properties	UIKit makes extensive use of properties throughout its class declarations. Properties were introduced to Mac OS X in version 10.5 and thus came along after the creation of many classes in the AppKit framework. Rather than simply mimic the same getter and setter methods in AppKit, properties are used in UIKit as a way to simplify the class interfaces. For information about how to use properties, see "Declared Properties" in *The Objective-C Programming Language*.
Controls and cells	Controls in UIKit do not use cells. Cells are used in Mac OS X as a lightweight alternative to views. Because views in UIKit are themselves very lightweight objects, cells are not needed. Despite the naming conventions, the cells designed for use with the `UITableView` class are actually based on the `UIView` class.

Difference	Discussion
Table views	The `UITableView` class in iOS can be thought of as a cross between the `NSTableView` and `NSOutlineView` 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 `UITableView` class displays a single column at a time and allows you to group related rows together into sections. It is also a means for displaying and editing hierarchical lists of information. For more information about the `UITableView` class, see *UITableView Class Reference*.
Menus	Nearly all applications written for iOS have a much smaller command set than does a comparable Mac OS X application, and so menubars 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, 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	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 by changing the desired property of the affected view. The only time you might need to use layers directly is when you need precise control over the layer tree or when you need features not exposed at the view level. For information about how Core Animation layers are integrated into the drawing model of iOS, see *View Programming Guide for iOS*.

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

Foundation Framework Differences

A version of the Foundation framework is available in both Mac OS X and 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. Table 6-2 lists some of the major areas of functionality that are not included in iOS, however, along with the reasons why the related classes are not available. Wherever possible, this table lists alternative technologies that you can use instead.

Table 6-2 Foundation technologies unavailable in iOS

Technology	Notes
Metadata and predicate management	Spotlight metadata and search predicates are not supported in iOS because Spotlight itself is not supported.
Distributed objects and port name server management	The Distributed Objects technology is not available, but you can still use the `NSPort` family of classes to interact with ports and sockets. You can also use the Core Foundation and CFNetwork frameworks to handle your networking needs.

Technology	Notes
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.
Objective-C garbage collection	Garbage collection is not supported in iOS. Instead, you must use the memory-managed model, whereby you retain objects to claim ownership and release objects when you no longer need them.
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 Mac OS X but not in iOS, see the class hierarchy diagram located in ""The Foundation Framework"" in *Foundation Framework Reference*.

Changes to Other Frameworks

Table 6-3 lists the key differences in other frameworks found in iOS.

Table 6-3 Differences in frameworks common to iOS and Mac OS X

Framework	Differences
`AddressBook.framework`	This framework contains the interfaces for accessing user contacts. Although it shares a same name, the iOS version of this framework is very different from its Mac OS X counterpart. 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. For more information, see *Address Book Framework Reference*.

`AudioToolbox.framework`
`AudioUnit.framework`
`CoreAudio.framework`
	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. For information on how to use the audio support, see Multimedia Support in *iOS Application Programming Guide*

`CFNetwork.framework`	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. For more information, see *CFNetwork Framework Reference*.

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

iOS Developer Tools

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

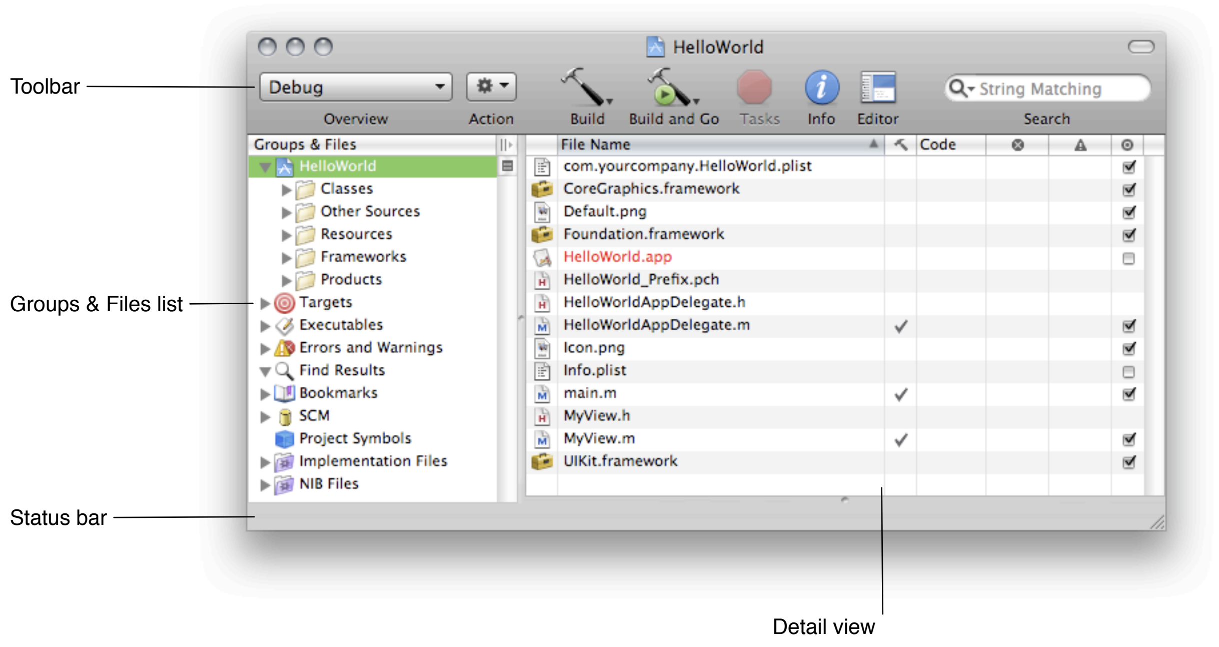
Xcode

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, build your code into an executable, and run and debug your code either in iPhone Simulator or on a device. Xcode incorporates a number of features to make developing 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 advanced documentation viewer for viewing and searching Apple documentation
- A context-sensitive inspector for viewing information about selected code symbols
- An advanced build system with dependency checking and build rule evaluation
- GCC compilers supporting C, C++, Objective-C, Objective-C++, and Objective-C 2.0, and other languages
- Integrated source-level debugging using GDB
- Distributed computing, enabling you to distribute large projects over several networked machines
- Predictive compilation that speeds single-file compile turnaround times
- Advanced debugging features such as fix and continue and custom data formatters
- Advanced refactoring tools that let you make global modifications to your code without changing its overall behavior
- Support for project snapshots, which provide a lightweight form of local source-code management
- Support for launching performance tools to analyze your software
- Support for integrated source-code management
- AppleScript support for automating the build process
- Support for DWARF and Stabs debugging information (DWARF debugging information is generated by default for all new projects)

To create a new 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 and Files list, you manage the files in your project, including the source files and build targets that are created from those source files. In the toolbar, you access commonly used tools and commands. And in the details pane, you can configure a space for working on your project. Other aspects of the project window provide you with contextual information about your project.

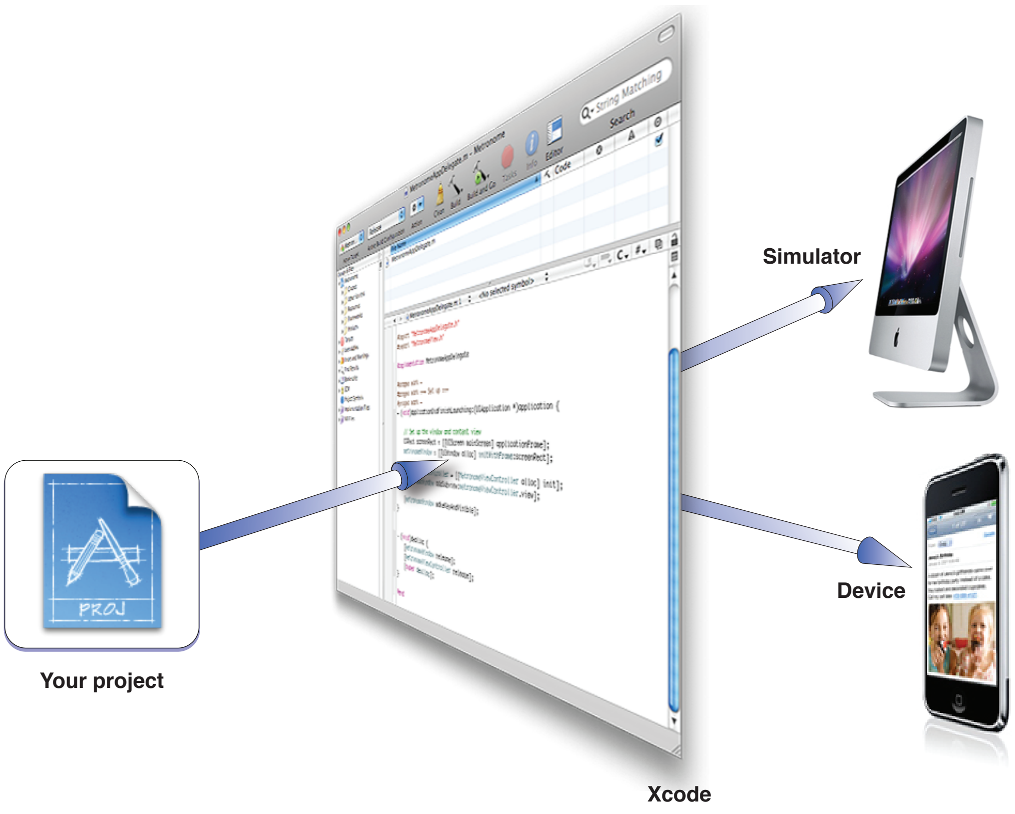
Figure A-1 An Xcode project window



File Name		Code			
com.yourcompany.HelloWorld.plist					✓
CoreGraphics.framework					✓
Default.png					✓
Foundation.framework					✓
HelloWorld.app					☐
HelloWorld_Prefix.pch					
HelloWorldAppDelegate.h					
HelloWorldAppDelegate.m	✓				✓
Icon.png					✓
Info.plist					☐
main.m	✓				✓
MyView.h					
MyView.m	✓				✓
UIKit.framework					✓

When you build your application in Xcode, you have a choice of building it for iPhone Simulator or for a device. The simulator provides a local environment for testing your applications to make sure they behave essentially the way you want. After you are satisfied with your application's basic behavior, you can tell Xcode to build your application and run it on an 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

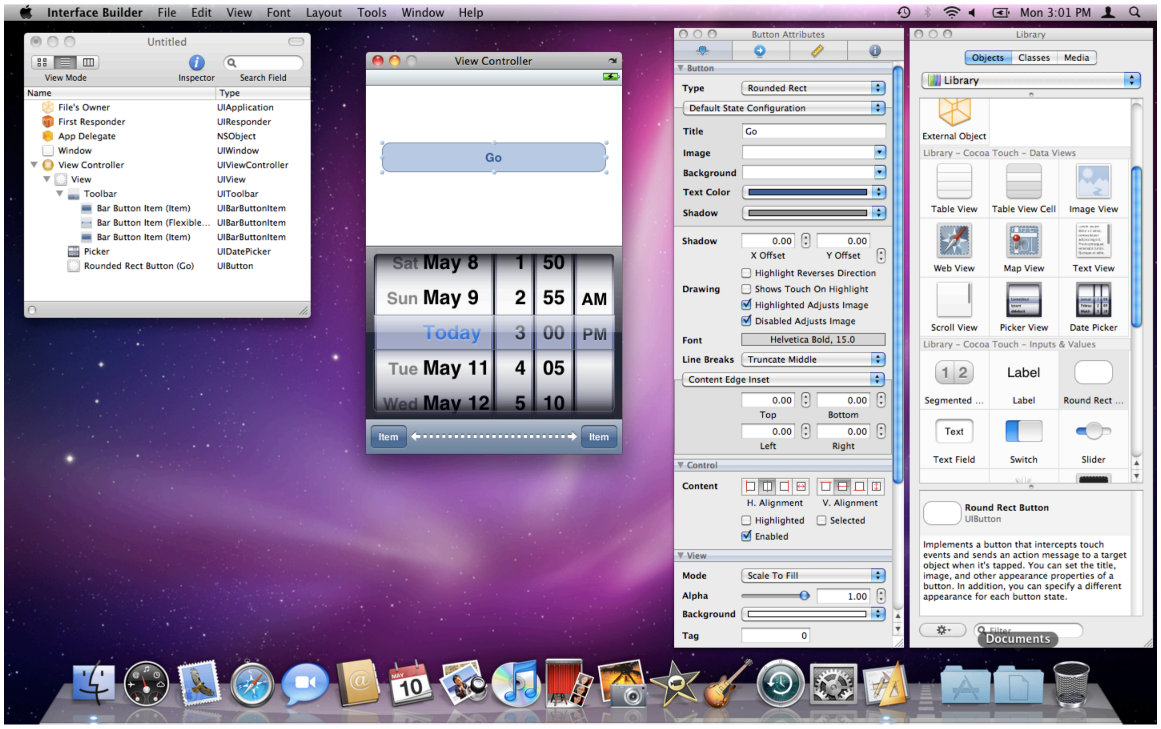


For details on how to build and run your project on iOS, see *iOS Development Guide*. For more information about the overall Xcode environment, see *A Tour of Xcode*.

Interface Builder

Interface Builder is the tool you use to assemble your application's user interface visually. Using Interface Builder, you assemble your application's window by dragging and dropping preconfigured components onto it, as shown in Figure A-3. The components include standard system controls such as switches, text fields, and buttons, and also custom views to represent the views your application provides. After you've placed the components on the window's surface, you can position them by dragging them around, configure their attributes using the inspector, and establish the relationships between those objects and your code. When your interface looks the way you want it, you save the contents to a nib file, which is a custom resource file format.

Figure A-3 Building iOS interfaces using Interface Builder



Name	Type
File's Owner	UIApplication
First Responder	UIResponder
App Delegate	NSObject
Window	UIWindow
View Controller	UIViewController
View	UIView
Toolbar	UIToolbar
Bar Button Item (Item)	UIBarButtonItem
Bar Button Item (Flexible...	UIBarButtonItem
Bar Button Item (Item)	UIBarButtonItem
Picker	UIDatePicker
Rounded Rect Button (Go)	UIButton

Button
Type: Rounded Rect
Default State Configuration
Title: Go
Image
Background
Text Color
Shadow
Shadow: 0.00 X Offset, 0.00 Y Offset
Drawing: [ ] Highlight Reverses Direction; [ ] Shows Touch On Highlight; [✓] Highlighted Adjusts Image; [✓] Disabled Adjusts Image
Font: Helvetica Bold, 15.0
Line Breaks: Truncate Middle
Content Edge Inset: 0.00 Top, 0.00 Bottom, 0.00 Left, 0.00 Right

Control
Content: H. Alignment, V. Alignment; [ ] Highlighted; [ ] Selected; [✓] Enabled

View
Mode: Scale To Fill
Alpha: 1.00
Background
Tag: 0

Round Rect Button
UIButton
Implements a button that intercepts touch events and sends an action message to a target object when it's tapped. You can set the title, image, and other appearance properties of a button. In addition, you can specify a different appearance for each button state.

The nib files you create in Interface Builder contain all the information that UIKit needs to recreate the same objects in your application at runtime. Loading a nib file creates runtime versions of all the objects stored in the file, configuring them exactly as they were in Interface Builder. It also uses the connection information you specified to establish connections between the newly created objects and any existing objects in your application. These connections provide your code with pointers to the nib-file objects and also provide the information the objects themselves need to communicate user actions to your code.

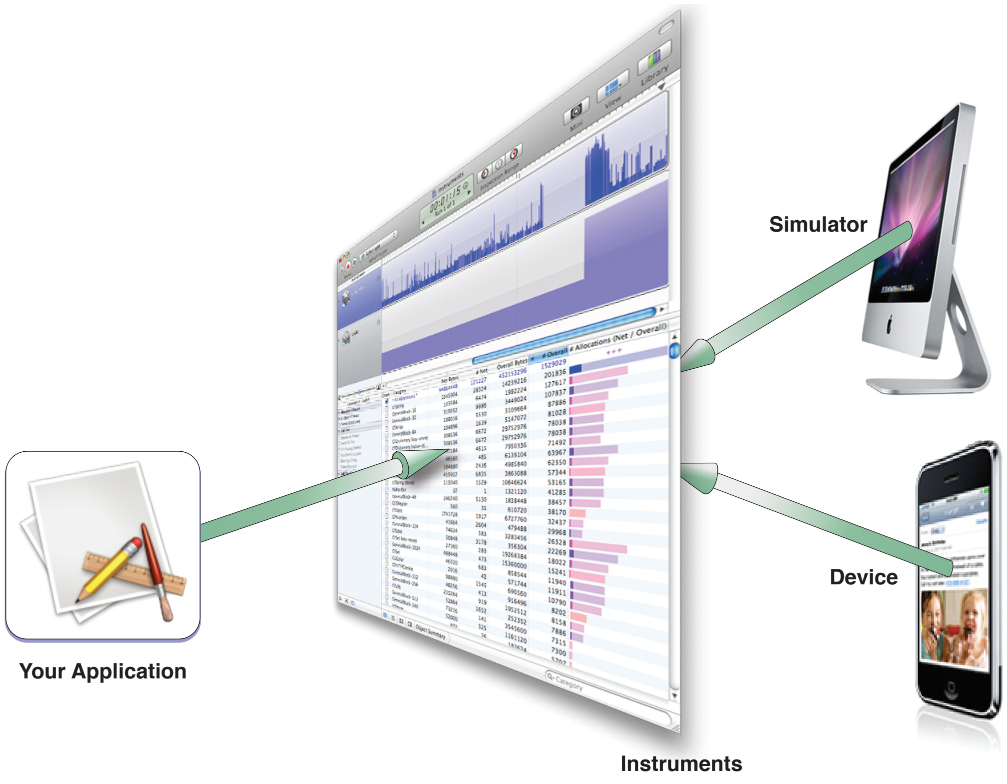
Overall, using Interface Builder saves a tremendous amount of time when it comes to creating your application's user interface. Interface Builder eliminates the custom code needed to create, configure, and position the objects that make up your interface. Because it is a visual editor, you get to see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

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

Instruments

To ensure that you deliver the best user experience for your software, the Instruments environment lets you analyze the performance of your iOS applications while running in the simulator or on a device. Instruments gathers data from your running application and presents that data in a graphical display called the timeline. You can gather data about your application's memory usage, disk activity, network activity, and graphics performance. The timeline view can display all of the different types of information side by side, letting you correlate the overall behavior of your application, not just the behavior in one specific area. To get even more detailed information, you can also view the detailed samples that Instruments gathers.

Figure A-4 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 *iOS Development Guide*. For general information on how to use Instruments, see *Instruments User Guide*.

Shark

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

For more information about using Shark with iOS–based devices, see *Shark 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 index list any key prefixes used by the classes, methods, functions, types, or constants of the framework. You should avoid using any of the specified prefixes in your own symbol names.

Device Frameworks

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

Table B-1 Device frameworks

Name	First available	Prefixes	Description
`Accelerate.framework`	4.0	`cblas`, `vDSP`	Contains accelerated math and DSP functions. See *Accelerate Framework Reference*.
`AddressBook.framework`	2.0	`AB`	Contains functions for accessing the user's contacts database directly. See *Address Book Framework Reference*.
`AddressBookUI.framework`	2.0	`AB`	Contains classes for displaying the system-defined people picker and editor interfaces. See *Address Book UI Framework Reference for iOS*.
`AssetsLibrary.framework`	4.0	`AL`	Contains classes for accessing the user's photos and videos. See *Assets Library Framework Reference*.
`AudioToolbox.framework`	2.0	`AU`, `Audio`	Contains the interfaces for handling audio stream data and for playing and recording audio. See *Audio Toolbox Framework Reference*.
`AudioUnit.framework`	2.0	`AU`, `Audio`	Contains the interfaces for loading and using audio units. See *Audio Unit Framework Reference*.
`AVFoundation.framework`	2.2	`AV`	Contains Objective-C interfaces for playing and recording audio. See *AV Foundation Framework Reference*.

Name	First available	Prefixes	Description
`CFNetwork.framework`	2.0	`CF`	Contains interfaces for accessing the network via the WiFi and cellular radios. See *CFNetwork Framework Reference*.
`CoreAudio.framework`	2.0	`Audio`	Provides the data types used throughout Core Audio. See *Core Audio Framework Reference*.
`CoreData.framework`	3.0	`NS`	Contains interfaces for managing your application's data model. See *Core Data Framework Reference*.
`CoreFoundation.framework`	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 Reference*.
`CoreGraphics.framework`	2.0	`CG`	Contains the interfaces for Quartz 2D. See *Core Graphics Framework Reference*.
`CoreLocation.framework`	2.0	`CL`	Contains the interfaces for determining the user's location. See *Core Location Framework Reference*.
`CoreMedia.framework`	4.0	`CM`	Contains low-level routines for manipulating audio and video. See *Core Media Framework Reference*.
`CoreMotion.framework`	4.0	`CM`	Contains interfaces for accessing accelerometer and gyro data. See *Core Motion Framework Reference*.
`CoreTelephony.framework`	4.0	`CT`	Contains routines for accessing telephony-related information. See *Core Telephony Framework Reference*.
`CoreText.framework`	3.2	`CT`	Contains a text layout and rendering engine. See *Core Text Reference Collection*.
`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 Reference*.
`EventKitUI.framework`	4.0	`EK`	Contains classes for displaying the standard system calendar interfaces. See *Event Kit UI Framework Reference*.
`External-Accessory.framework`	3.0	`EA`	Contains interfaces for communicating with attached hardware accessories. See *External Accessory Framework Reference*.
`Foundation.framework`	2.0	`NS`	Contains the classes and methods for the Cocoa Foundation layer. See *Foundation Framework Reference*.

Name	First available	Prefixes	Description
`GameKit.framework`	3.0	`GK`	Contains the interfaces for managing peer-to-peer connectivity. See *Game Kit Framework Reference*.
`iAd.framework`	4.0	`AD`	Contains classes for displaying advertisements in your application. See *iAd Framework Reference*.
`ImageIO.framework`	4.0	`CG`	Contains classes for reading and writing image data. See *Image I/O Reference Collection*.
`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 looking up reverse geocoding coordinates. See *Map Kit Framework Reference*.
`MediaPlayer.framework`	2.0	`MP`	Contains interfaces for playing full-screen video. See *Media Player Framework Reference*.
`MessageUI.framework`	3.0	`MF`	Contains interfaces for composing and queuing email messages. See *Message UI Framework Reference*.
`MobileCore-Services.framework`	3.0	`UT`	Defines the uniform type identifiers (UTIs) supported by the system.
`OpenAL.framework`	2.0	`AL`	Contains the interfaces for OpenAL, a cross-platform positional audio library. For more information, go to http://www.openal.org.
`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 Reference*.
`QuartzCore.framework`	2.0	`CA`	Contains the Core Animation interfaces. See *Quartz Core Framework Reference*.
`QuickLook.framework`	4.0	`QL`	Contains interfaces for previewing files. See *Quick Look Framework Reference*.
`Security.framework`	2.0	`CSSM`, `Sec`	Contains interfaces for managing certificates, public and private keys, and trust policies. See *Security Framework Reference*.
`StoreKit.framework`	3.0	`SK`	Contains interfaces for handling the financial transactions associated with in app purchases. See *Store Kit Framework Reference*.

Name	First available	Prefixes	Description
`System-Configuration.framework`	2.0	`SC`	Contains interfaces for determining the network configuration of a device. See *System Configuration Framework Reference*.
`UIKit.framework`	2.0	`UI`	Contains classes and methods for the iOS application user-interface layer. See *UIKit Framework Reference*.

Simulator Frameworks

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

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

iOS uses symbolic links to point to the most current version of most libraries. When linking to a dynamic shared library, use the symbolic link instead of a link to a specific version of the library. Library versions may change in future versions of 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
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.