iOS Technology Overview

Developer

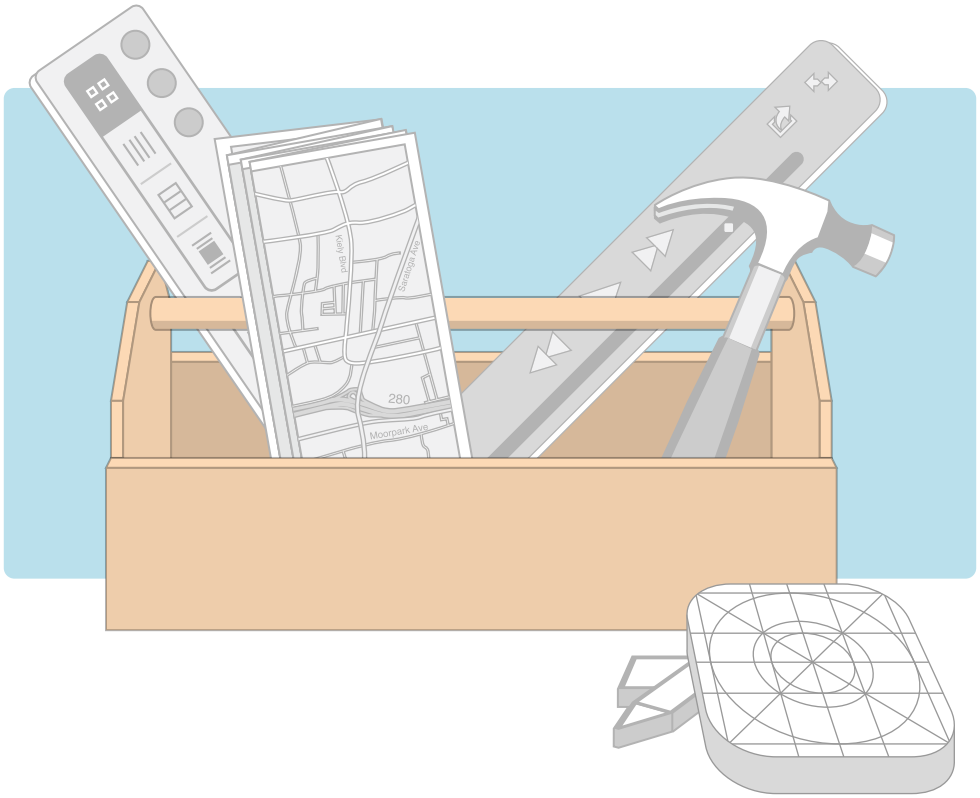
Contents

Figures and Tables

About the iOS Technologies

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

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



Note: It is possible to create web apps using a combination of HTML, cascading style sheets (CSS), and JavaScript code. Web apps run inside the Safari web browser and require a network connection to access your web server. This document does not cover the creation of web apps. For more information about creating web apps in Safari, see *Safari Web Content Guide*.

At a Glance

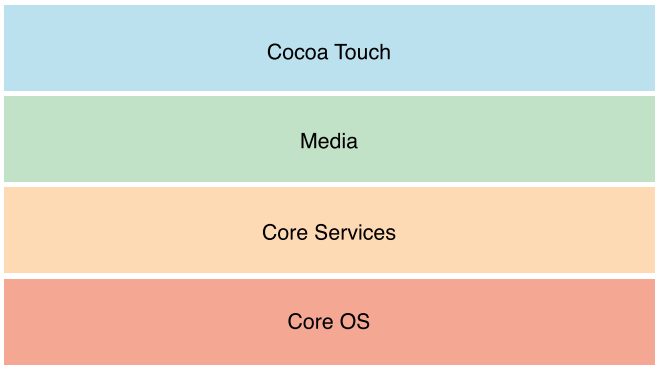
The iOS SDK provides the resources you need to develop native iOS apps. Understanding a little about the technologies and tools contained in the SDK can help you make better choices about how to design and implement your apps.

The iOS Architecture Is Layered

At the highest level, iOS acts as an intermediary between the underlying hardware and the apps you create. Apps do not talk to the underlying hardware directly. Instead, they communicate with the hardware through a set of well-defined system interfaces. These interfaces make it easy to write apps that work consistently on devices having different hardware capabilities.

The implementation of iOS technologies can be viewed as a set of layers, which are shown in Figure I-1. Lower layers contain fundamental services and technologies. Higher-level layers build upon the lower layers and provide more sophisticated services and technologies.

Figure I-1 Layers of iOS



As you write your code, it is recommended that you prefer the use of higher-level frameworks over lower-level frameworks whenever possible. The higher-level frameworks are there to provide object-oriented abstractions for lower-level constructs. These abstractions generally make it much easier to write code because they reduce

the amount of code you have to write and encapsulate potentially complex features, such as sockets and threads. You may use lower-level frameworks and technologies, too, if they contain features not exposed by the higher-level frameworks.

Relevant chapters: Cocoa Touch Layer (page 11), Media Layer (page 22), Core Services Layer (page 35), Core OS Layer (page 48)

The iOS Technologies Are Packaged as Frameworks

Apple delivers most of its system interfaces in special packages called frameworks. A **framework** is a directory that contains a dynamic shared library and the resources (such as header files, images, and helper apps) needed to support that library. To use frameworks, you add them to your app project from Xcode.

Relevant sections: iOS Frameworks (page 52)

The Developer Library Is There to Help You

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

You can access the iOS Developer Library from the Apple Developer website or from Xcode. In Xcode, choose Help > Documentation and API Reference to display the Xcode documentation window, which is the central resource for accessing information about iOS development. Use the documentation window to browse, search, and bookmark documents.

How to Use This Document

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

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

This document does not provide information about user-level features that have no impact on the software development process, nor does it list the hardware capabilities of specific iOS devices. New developers should find this document useful for getting familiar with iOS. Experienced developers can use it as a road map for exploring specific technologies and development techniques.

See Also

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

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

To develop on a device, you sign up for Apple's paid iOS Developer program and then configure a device for development purposes. After you sign up, obtain a copy of Xcode and the iOS SDK at the iOS Dev Center

Cocoa Touch Layer

The Cocoa Touch layer contains key frameworks for building iOS apps. These frameworks define the appearance of your app. They also provide the basic app infrastructure and support for key technologies such as multitasking, touch-based input, push notifications, and many high-level system services. When designing your apps, you should investigate the technologies in this layer first to see if they meet your needs.

High-Level Features

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

App Extensions

iOS 8 lets you extend select areas of the system by supplying an **app extension**, which is code that enables custom functionality within the context of a user task. For example, you might supply an app extension that helps users post content to your social sharing website. After users install and enable this extension, they can choose it when they tap the Share button in their current app. Your custom sharing extension provides the code that accepts, validates, and posts the user's content. The system lists the extension in the sharing menu and instantiates it when the user chooses it.

In Xcode, you create an app extension by adding a preconfigured app extension target to an app. After a user installs an app that contains an extension, the extension can be enabled by the user in the Settings app. When the user is running other apps, the system makes the enabled extension available in the appropriate system UI, such as the Share menu.

iOS supports app extensions for the following areas, which are known as extension points:

- **Share**. Share content with social websites or other entities.
- **Action**. Perform a simple task with the current content.
- **Widget**. Provide a quick update or enable a brief task in the Today view of Notification Center.
- **Photo editing**. Perform edits to a photo or video within the Photos app.
- **Document provider**. Provide a document storage location that can be accessed by other apps. Apps that use a document picker view controller can open files managed by the Document Provider or move files into the Document Provider.

- **Custom keyboard**. Provide a custom keyboard that the user can choose in place of the system keyboard for all apps on the device.

Each extension point defines appropriate APIs for its purposes. When you use an app extension template to begin development, you get a default target that contains method stubs and property list settings defined by the extension point you chose.

For more information on creating extensions, see *App Extension Programming Guide*.

Handoff

Handoff is a feature in OS X and iOS that extends the user experience of continuity across devices. Handoff enables users to begin an activity on one device, then switch to another device and resume the same activity on the other device. For example, a user who is browsing a long article in Safari moves to an iOS device that's signed into the same Apple ID, and the same webpage automatically opens in Safari on iOS, with the same scroll position as on the original device. Handoff makes this experience as seamless as possible.

To participate in Handoff, an app adopts a small API in Foundation. Each ongoing activity in an app is represented by a user activity object that contains the data needed to resume an activity on another device. When the user chooses to resume that activity, the object is sent to the resuming device. Each user activity object has a delegate object that is invoked to refresh the activity state at opportune times, such as just before the user activity object is sent between devices.

If continuing an activity requires more data than is easily transferred by the user activity object, the resuming app has the option to open a stream to the originating app. Document-based apps automatically support activity continuation for users working with iCloud-based documents.

For more information about how to support Handoff, see *Handoff Programming Guide*.

Document Picker

The document picker view controller (`UIDocumentPickerViewController`) grants users access to files outside your application's sandbox. It is a simple mechanism for sharing documents between apps. It also enables more complex workflows, because users can edit a single document with multiple apps.

The document picker lets you access files from a number of document providers. For example, the iCloud document provider grants access to documents stored inside another app's iCloud container. Third-party developers can provide additional document providers by using the Storage Provider extension.

For more information about how to use the document picker, see the *Document Picker Programming Guide*.

AirDrop

AirDrop lets users share photos, documents, URLs, and other kinds of data with nearby devices. Support for sending files to other iOS devices using AirDrop is built into the existing `UIActivityViewController` class. This class displays different options for sharing the content that you specify. If you are not yet using this class, you should consider adding it to your interface.

To receive files sent using AirDrop, your app must do the following:

- Declare support for the appropriate document types in Xcode. (Xcode adds the appropriate keys to your app's `Info.plist` file.) The system uses this information to determine whether your app can open a given file.
- Implement the `application:openURL:sourceApplication:annotation:` method in your app delegate. (The system calls this method when a new file is received.)

Files sent to your app are placed in the `Documents/Inbox` directory of your app's home directory. If you plan to modify the file, you must move it out of this directory before doing so. (The system allows your app to read and delete files in this directory only.) Files stored in this directory are encrypted using data protection, so you must be prepared for the file to be inaccessible if the device is currently locked.

For more information about using an activity view controller to share data, see *UIActivityViewController Class Reference* .

TextKit

TextKit is a full-featured, high-level set of classes for handling text and fine typography. Using TextKit, you can lay out styled text into paragraphs, columns, and pages; you can flow text around arbitrary regions such as graphics; and you can use it to manage multiple fonts. If you were considering using Core Text to implement text rendering, you should consider TextKit instead. TextKit is integrated with all UIKit text-based controls to enable apps to create, edit, display, and store text more easily—and with less code than was previously possible in iOS.

TextKit comprises new UIKit classes, along with extensions to existing classes, including the following:

- The `NSAttributedString` class has been extended to support new attributes.
- The `NSLayoutManager` class generates glyphs and lays out text.
- The `NSTextContainer` class defines a region where text is laid out.
- The `NSTextStorage` class defines the fundamental interface for managing text-based content.

For more information about TextKit, see *Text Programming Guide for iOS* .

UIKit Dynamics

Apps can now specify dynamic behaviors for `UIView` objects and for other objects that conform to the `UIDynamicItem` protocol. (Objects that conform to this protocol are called *dynamic items*.) Dynamic behaviors offer a way to improve the user experience of your app by incorporating real-world behavior and characteristics into your app's user interface. UIKit dynamics supports the following types of behaviors:

- A `UIAttachmentBehavior` object specifies a connection between two dynamic items or between an item and a point. When one item (or point) moves, the attached item also moves. The connection is not completely static, though. An attachment behavior has damping and oscillation properties that determine how the behavior changes over time.
- A `UICollisionBehavior` object lets dynamic items participate in collisions with each other and with the behavior's specified boundaries. The behavior also lets those items respond appropriately to collisions.
- A `UIGravityBehavior` object specifies a gravity vector for its dynamic items. Dynamic items accelerate in the vector's direction until they collide with other appropriately configured items or with a boundary.
- A `UIPushBehavior` object specifies a continuous or instantaneous force vector for its dynamic items.
- A `UISnapBehavior` object specifies a snap point for a dynamic item. The item snaps to the point with a configured effect. For example, a dynamic item can snap to the point as if it were attached to a spring.

Dynamic behaviors become active when you add them to an animator object, which is an instance of the `UIDynamicAnimator` class. The animator provides the context in which dynamic behaviors execute. A given dynamic item can have multiple behaviors, but all of those behaviors must be animated by the same animator object.

For information about the behaviors you can apply, see *UIKit Framework Reference*.

Multitasking

Battery life is an important consideration for users of iOS devices and the multitasking model in iOS is designed to maximize battery life while giving apps the time they need to do critical work. When the user presses the Home button, the foreground app shifts to a background execution context. If the app has no more work to do, it is suspended from active execution and put into a "freeze-dried" state, where it remains in memory but does not execute any code. Apps that do need specific types of work can ask the system for background execution time. For example:

- An app can request a finite amount of time to complete some important task.
- An app that supports specific services (such as audio playback) can request time to provide those services.
- An app can use local notifications to generate user alerts at designated times, whether or not the app is running.

- An app can download content periodically from the network.
- An app can download content in response to a push notification.

For information on how to support the iOS multitasking model, see *App Programming Guide for iOS*.

Auto Layout

Auto layout helps you build dynamic interfaces with very little code. Using Auto Layout, you define rules for how to lay out the elements in your user interface. These rules express a larger class of relationships and are more intuitive to use than the springs and struts model that was used previously. For example, you can specify that a button always be 20 points from the left edge of its parent view.

The entities used in Auto Layout are Objective-C objects called *constraints*. Constraints provide several benefits:

- They support localization through the swapping of strings alone, instead of requiring you to update your layouts.
- They support mirroring of user interface elements for right-to-left languages, such as Hebrew and Arabic.
- They promote a better separation of responsibilities between objects in the view and controller layers.

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

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

Storyboards

Storyboards are the recommended way to design your app's user interface. Storyboards let you design your entire user interface in one place so that you can see all of your views and view controllers and understand how they work together. An important part of storyboards is the ability to define segues, which are transitions from one view controller to another. These transitions allow you to capture the flow of your user interface in addition to the content. You can define these transitions visually, in Xcode, or initiate them programmatically.

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

For more information about using storyboards to design your interface, see *Xcode Overview*. For information about how to access storyboards from your code, see the *UIStoryboard Class Reference*.

UI State Preservation

State preservation provides a seamless experience for users by having your app appear to be always running, even when it was not. If the system encounters memory pressure, it may be forced to terminate one or more background apps quietly. When an app moves from the foreground to the background, it can preserve the state of its views and view controllers. During its next launch cycle, it can use that preserved state information to restore its views and view controllers to their previous configurations and make it appear as if the app had never quit.

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

Apple Push Notification Service

Apple Push Notification service provides a way to alert users about new information, even when your app is not actively running. Using this service, you can push text notifications, add a badge to your app icon, or trigger audible alerts on user devices at any time. These messages let users know that they should open your app to receive the related information. In iOS 7, you can even push silent notifications to let your app know that new content is available to download.

From a design standpoint, there are two parts to making push notifications work for iOS apps. First, the app must request the delivery of notifications and process the notification data once it is delivered. Second, you need to provide a server-side process to generate the notifications in the first place. This process lives on your own local server and works with Apple Push Notification Service to trigger the notifications.

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

Local Notifications

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

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

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

Gesture Recognizers

Gesture recognizers detect common types of gestures, such as swipes and pinches in your app's views. Because they use the same heuristics as the system for detecting gestures, gesture recognizers offer a consistent behavior for your apps. To use one, you attach the gesture recognizer to your view and give it an action method to perform when the gesture occurs. The gesture recognizer does the difficult work of tracking the raw touch events and determining when they constitute the intended gesture.

All gesture recognizers are based on the `UIGestureRecognizer` class, which defines the basic behavior. UIKit supplies standard gesture recognizer subclasses to detect taps, pinches, pans, swipes, rotations. You can also tailor the behavior of most gesture recognizers to your app's needs. For example, you can tell a tap gesture recognizer to detect a specific number of taps before calling your action method.

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

Standard System View Controllers

Many system frameworks define view controllers for standard system interfaces. Whenever possible, use the provided view controllers rather than create your own. You are encouraged to use these view controllers in your apps 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 EventKit UI framework.
- **Compose an email or SMS message.** Use the view controllers in the Message UI framework.
- **Open or preview the contents of a file.** Use the `UIDocumentInteractionController` class in the UIKit framework.
- **Take a picture or choose a photo from the user's photo library.** Use the `UIImagePickerController` class in the UIKit framework.
- **Shoot a video clip.** Use the `UIImagePickerController` class in the UIKit framework.

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

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 app and also makes sure that your app uses the same interfaces as other apps, 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*.

EventKit UI Framework

The EventKit UI framework (`EventKitUI.framework`) provides view controllers for presenting the standard system interfaces for viewing and editing calendar-related events. This framework builds upon the event-related data in the EventKit framework, which is described in Event Kit Framework (page 43).

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

GameKit Framework

The GameKit framework (`GameKit.framework`) implements support for Game Center, which lets users share their game-related information online. Game Center provides support for the following features:

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

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

iAd Framework

The iAd framework (`iAd.framework`) lets you deliver banner-based advertisements from your app. 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 iAd Service to automatically handle all the work associated with loading and presenting rich media ads and responding to taps in those ads.

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

MapKit Framework

The MapKit framework (`MapKit.framework`) provides a scrollable map that you can incorporate into your app's user interface. Beyond just displaying a map, you can use the framework interfaces to customize the map's content and appearance. You can flag points of interest using annotations, and you can use custom overlays to intersperse your own content with the map content. For example, you might use an overlay to draw a bus route, or use annotations to highlight nearby shops and restaurants.

In addition to displaying maps, the MapKit framework integrates with the Maps app and Apple's map servers to facilitate directions. From the Maps app, users can delegate the providing of directions to any app that supports directions. Apps that provide specialized types of directions, such as subway routes, can register to provide those directions when asked. Apps can also request walking and driving directions from Apple servers and merge that route information with their custom directions to provide a complete point-to-point experience for the user.

For more information about using classes of the MapKit framework, see *Location and Maps Programming Guide* and *MapKit Framework Reference*.

Message UI Framework

The Message UI framework (`MessageUI.framework`) provides support for composing email or SMS messages from your app. The composition support consists of a view controller interface that you present in your app. You can populate the fields of this view controller to set the recipients, subject, body content, and any attachments you want to include with the message. After presenting the view controller, the user then has the option of editing the message before sending it.

For more information about the classes of the Message UI framework, see *Message UI Framework Reference*. For information about using the classes of this framework, see *System Messaging Programming Topics for iOS*.

Notification Center Framework

The Notification Center framework (`NotificationCenter.framework`) provides support for creating widgets that display information in Notification Center. For information about how to create notification center widgets, see *App Extension Programming Guide* and *Notification Center Framework Reference*.

PushKit Framework

The PushKit framework (PushKit.framework) provides registration support for VoIP apps. This framework replaces the previous APIs for registering VoIP apps. Instead of keeping a persistent connection open, and thus draining the device's battery, an app can use this framework to receive push notifications when there is an incoming call.

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

Twitter Framework

The Twitter framework (`Twitter.framework`) has been replaced by the Social framework, which supports a UI for generating tweets and support for creating URLs to access the Twitter service. For more information, see Social Framework (page 46).

UIKit Framework

The UIKit framework (`UIKit.framework`) provides crucial infrastructure for implementing graphical, event-driven apps in iOS, including the following:

- Basic app management and infrastructure, including the app's main run loop
- User interface management, including support for storyboards and nib files
- A view controller model to encapsulate the contents of your user interface
- Objects representing the standard system views and controls
- Support for handling touch- and motion-based events
- Support for a document model that includes iCloud integration; see *Document-Based App Programming Guide for iOS*
- Graphics and windowing support, including support for external displays; see *View Programming Guide for iOS*
- Multitasking support; see Multitasking (page 14)
- Printing support; see *Drawing and Printing Guide for iOS*
- Support for customizing the appearance of standard UIKit controls

- Support for text and web content
- Cut, copy, and paste support
- Support for animating user-interface content
- Integration with other apps on the system through URL schemes and framework interfaces
- Accessibility support for disabled users
- Support for the Apple Push Notification service; see Apple Push Notification Service (page 16)
- Local notification scheduling and delivery; see Local Notifications (page 16)
- PDF creation
- Support for using custom input views that behave like the system keyboard
- Support for creating custom text views that interact with the system keyboard
- Support for sharing content through email, Twitter, Facebook, and other services

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

- The built-in camera (where present)
- The user's photo library
- Device name and model information
- Battery state information
- Proximity sensor information
- Remote control information from attached headsets

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

Media Layer

The Media layer contains the graphics, audio, and video technologies you use to implement multimedia experiences in your apps. The technologies in this layer make it easy for you to build apps that look and sound great.

Graphics Technologies

High-quality graphics are an important part of all apps, and iOS provides numerous technologies to help put your custom art and graphics onscreen. The iOS graphics technologies offer a wide range of support, working seamlessly with the UIKit view architecture to make it easy to deliver content. You can use the standard views to deliver a high-quality interface quickly, or you can create your own custom views and use any of the technologies listed in Table 2-1 to deliver an even richer graphical experience.

Table 2-1 Graphics technologies in iOS

Technology	Description
UIKit graphics	UIKit defines high-level support for drawing images and Bézier paths and for animating the content of your views. In addition to providing classes to implement drawing support, UIKit views provide a fast and efficient way to render images and text-based content. Views can also be animated, both explicitly and using UIKit dynamics, to provide feedback and promote user interactivity. For more information about the classes of the UIKit framework, see *UIKit Framework Reference*.
Core Graphics framework	**Core Graphics** (also known as *Quartz*) is the native drawing engine for iOS apps and provides support for custom 2D vector- and image-based rendering. Although not as fast as OpenGL ES rendering, this framework is well suited for situations where you want to render custom 2D shapes and images dynamically. For more information, see Core Graphics Framework (page 29).
Core Animation	**Core Animation** (part of the Quartz Core framework) is a foundational technology that optimizes the animation experience of your apps. UIKit views use Core Animation to provide view-level animation support. You can use Core Animation directly when you want more control over the behavior of your animations. For more information, see Quartz Core Framework (page 33).

Technology	Description
Core Image	**Core Image** provides advanced support for manipulating video and still images in a nondestructive manner. For more information, see Core Image Framework (page 29).
OpenGL ES and GLKit	**OpenGL ES** handles advanced 2D and 3D rendering using hardware-accelerated interfaces. This framework is traditionally used by game developers, or by anyone wanting to implement an immersive graphical experience. This framework gives you full control over the rendering process and offers the frame rates needed to create smooth animations. For more information, see OpenGL ES Framework (page 32). **GLKit** is a set of Objective-C classes that provide the power of OpenGL ES using an object-oriented interface. For more information, see GLKit Framework (page 30).
Metal	**Metal** provides extremely low-overhead access to the A7 GPU, enabling incredibly high performance for your sophisticated graphics rendering and computation tasks. Metal eliminates many performance bottlenecks—such as costly state validation—that are found in traditional graphics APIs. For more information, see Metal Framework (page 31).
TextKit and Core Text	**TextKit** is a family of UIKit classes used to perform fine typography and text management. If your app performs advanced text manipulations, TextKit provides seamless integration with the rest of your views. For more information, see Text Kit (page 13). **Core Text** is a lower-level C-based framework for handling advanced typography and layout. For more information, see Core Text Framework (page 29).
Image I/O	**Image I/O** provides interfaces for reading and writing most image formats. For more information, see Image I/O Framework (page 30).
Photos Library	The **Photos** and PhotosUI frameworks provide access to a user's photos, videos, and media. You use this framework in places where you want to integrate the user's own content with your app. For more information, see Photos Framework (page 32) and Photos UI Framework (page 33).

iOS provides built-in support for apps running on either Retina displays or standard-resolution displays. For vector-based drawing, the system frameworks automatically use the extra pixels of a Retina display to improve the crispness of your content. And if you use images in your app, UIKit provides support for loading high-resolution variants of your existing images automatically. For more information about what you need to do to support high-resolution screens, see App-Related Resources in *App Programming Guide for iOS*.

Audio Technologies

The iOS audio technologies work with the underlying hardware to provide a rich audio experience for your users. This experience includes the ability to play and record high-quality audio, to handle MIDI content, and to work with a device's built-in sounds.

If your app uses audio, there are several technologies available for you to use. Table 2-2 lists these frameworks and describes the situations where you might use each.

Table 2-2 Audio technologies in iOS

Technology	Description
Media Player framework	This high-level framework provides easy access to a user's iTunes library and support for playing tracks and playlists. Use this framework when you want to integrate audio into your app quickly and when you don't need to control playback behavior. For more information, see Media Player Framework (page 31).
AV Foundation	**AV Foundation** is an Objective-C interface for managing the recording and playback of audio and video. Use this framework for recording audio and when you need fine-grained control over the audio playback process. For more information, see AV Foundation Framework (page 27).
OpenAL	**OpenAL** is an industry-standard technology for delivering positional audio. Game developers frequently use this technology to deliver high-quality audio using a set of cross-platform interfaces. For more information, see OpenAL Framework (page 32).
Core Audio	**Core Audio** is a set of frameworks that provide both simple and sophisticated interfaces for the recording and playback of audio and MIDI content. This framework is for advanced developers who need fine-grained control over their audio. For more information, see Core Audio (page 28).

iOS supports many industry-standard and Apple-specific audio formats, including the following:

- AAC
- Apple Lossless (ALAC)
- A-law
- IMA/ADPCM (IMA4)
- Linear PCM
- µ-law
- DVI/Intel IMA ADPCM

- Microsoft GSM 6.10
- AES3-2003

Video Technologies

The iOS video technologies provide support for managing static video content in your app or playing back streaming content from the Internet. For devices with the appropriate recording hardware, you can also record video and incorporate it into your app. Table 2-3 lists the technologies that support video playback and recording.

Table 2-3 Video technologies in iOS

Technology	Description
`UIImagePickerController`	The `UIImagePickerController` class is a UIKit view controller for choosing user media files. You can use this view controller to prompt the user for an existing picture or video or to let the user capture new content. For more information, see *Camera Programming Topics for iOS*.
AVKit	The AVKit framework provides a set of simple-to-use interfaces for presenting video. This framework supports both full-screen and partial-screen video playback and supports optional playback controls for the user. For more information, see AVKit Framework (page 27).
AV Foundation	AV Foundation provides advanced video playback and recording capabilities. Use this framework in situations where you need more control over the presentation or recording of video. For example, augmented reality apps could use this framework to layer live video content with other app-provided content. For more information, see AV Foundation Framework (page 27).
Core Media	The Core Media framework defines the low-level data types and interfaces for manipulating media. Most apps do not need to use this framework directly, but it is available when you need unparalleled control over your app's video content. For more information, see Core Media Framework (page 42).

iOS supports many industry-standard video formats and compression standards, including the following:

- H.264 video, up to 1.5 Mbps, 640 by 480 pixels, 30 frames per second, Low-Complexity version of the H.264 Baseline Profile with AAC-LC audio up to 160 Kbps, 48 kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats
- H.264 video, up to 768 Kbps, 320 by 240 pixels, 30 frames per second, Baseline Profile up to Level 1.3 with AAC-LC audio up to 160 Kbps, 48 kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats

- MPEG-4 video, up to 2.5 Mbps, 640 by 480 pixels, 30 frames per second, Simple Profile with AAC-LC audio up to 160 Kbps, 48 kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats
- Numerous audio formats, including the ones listed in Audio Technologies (page 24)

AirPlay

AirPlay lets your app stream audio and video content to Apple TV and stream audio content to third-party AirPlay speakers and receivers. AirPlay support is built into numerous frameworks—UIKit framework, Media Player framework, AV Foundation framework, and the Core Audio family of frameworks—so in most cases you do not need to do anything special to support it. Any content you play using these frameworks is automatically made eligible for AirPlay distribution. When the user chooses to play your content using AirPlay, it is routed automatically by the system.

Additional options for delivering content over AirPlay include the following:

- To extend the content displayed by an iOS device, create a second window object and assign it to any `UIScreen` objects that are connected to the device through AirPlay. Use this technique when the content you display on the attached screen is different than the content displayed on the iOS device.
- The playback classes of the Media Player framework automatically support AirPlay. You can also display Now Playing content on a connected Apple TV using AirPlay.
- Use the `AVPlayer` class in AV Foundation to manage your app's audio and video content. This class supports streaming its content via AirPlay when enabled by the user.
- For web-based audio and video, you can allow that content to be played over AirPlay by including an `embed` tag with the `airplay` attribute. The `UIWebView` class also supports media playback using AirPlay.

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

Media Layer Frameworks

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

Assets Library Framework

The Assets Library framework (`AssetsLibrary.framework`) provides access to the photos and videos managed by the Photos app on a user's device. Use this framework to access items in the user's saved photos album or in any albums 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

The AV Foundation framework (`AVFoundation.framework`) provides a set of Objective-C classes for playing, recording, and managing audio and video content. Use this framework when you want to integrate media capabilities seamlessly into your app's user interface. You can also use it for more advanced media handling. For example, you use this framework to play multiple sounds simultaneously and control numerous aspects of the playback and recording process.

The services offered by this framework include:

- Audio session management, including support for declaring your app's audio capabilities to the system
- Management of your app's media assets
- Support for editing media content
- The ability to capture audio and video
- The ability to play back audio and video
- 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
- Support for streaming content over AirPlay

For more information about how to use AV Foundation, see *AV Foundation Programming Guide*. For information about the classes of the AV Foundation framework, see *AV Foundation Framework Reference*.

AVKit Framework

The AVKit framework (`AVKit.framework`) leverages existing objects in AV Foundation to manage the presentation of video on a device. It is intended as a replacement for the Media Player framework when you need to display video content.

For more information about this framework, see the header files.

Core Audio

Core Audio is a family of frameworks (listed in Table 2-4) that provides native support for handling audio. These frameworks support the generation, recording, mixing, and playing of audio in your apps. You can also use these interfaces to work with MIDI content and to stream audio and MIDI content to other apps.

Table 2-4 Core Audio frameworks

Framework	Services
`CoreAudio.framework`	Defines the audio data types used throughout Core Audio. For more information, see *Core Audio Framework Reference*.
`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. For more information, see *Audio Toolbox Framework Reference*.
`AudioUnit.framework`	Provides services for using the built-in audio units, which are audio processing modules. This framework also supports vending your app's audio content as an audio component that is visible to other apps. For more information, see *Audio Unit Framework Reference*.
`CoreMIDI.framework`	Provides a standard way to communicate with MIDI devices, including hardware keyboards and synthesizers. You use this framework to send and receive MIDI messages and to interact with MIDI peripherals connected to an iOS-based device using the dock connector or network. For more information, see *Core MIDI Framework Reference*.
`MediaToolbox.framework`	Provides access to the audio tap interfaces.

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

CoreAudioKit Framework

The CoreAudioKit framework (`CoreAudioKit.framework`) provides standard views for managing connections between apps that support inter-app audio. One view provides a switcher that displays the icons of other connected apps and the other view displays the transport controls that the user can use to manipulate the audio provided by the host app.

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

Core Graphics Framework

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

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

Core Image Framework

The Core Image framework (`CoreImage.framework`) provides a powerful set of built-in filters for manipulating video and still images. You can use the built-in filters for everything from touching up and correcting photos to face, feature, and QR code detection. The advantage of these filters is that they operate in a nondestructive manner, leaving your original images unchanged. Because the filters are optimized for the underlying hardware, they are fast and efficient.

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

Core Text Framework

The Core Text framework (`CoreText.framework`) offers a simple, high-performance C-based interface for laying out text and handling fonts. This framework is for apps that do not use TextKit but that still want the kind of advanced text handling capabilities found in word processor apps. The framework provides a sophisticated text layout engine, including the ability to wrap text around other content. It also supports advanced text styling using multiple fonts and rendering attributes.

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

Core Video Framework

The Core Video framework (`CoreVideo.framework`) provides buffer and buffer-pool support for the Core Media framework (described in Core Media Framework (page 42)). Most apps never need to use this framework directly.

Game Controller Framework

The Game Controller framework (`GameController.framework`) lets you discover and configure Made-for-iPhone/iPod/iPad (MFi) game controller hardware in your app. Game controllers can be devices connected physically to an iOS device or connected wirelessly over Bluetooth. The Game Controller framework notifies your app when controllers become available and lets you specify which controller inputs are relevant to your app.

For more information about supporting game controllers, *Game Controller Programming Guide*.

GLKit Framework

The GLKit framework (`GLKit.framework`) contains a set of Objective-C based utility classes that simplify the effort required to create an OpenGL ES app. GLKit supports four key areas of app development:

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

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

Image I/O Framework

The Image I/O framework (`ImageIO.framework`) provides interfaces for importing and exporting image data and image metadata. This framework makes use of the Core Graphics data types and functions and supports all of the standard image types available in iOS. You can also use this framework to access Exif and IPTC metadata properties for images.

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

Media Accessibility Framework

The Media Accessibility framework (`MediaAccessibility.framework`) manages the presentation of closed-caption content in your media files. This framework works in conjunction with new settings that let the user enable the display of closed captions.

For information about the contents of this framework, see the header files.

Media Player Framework

The Media Player framework (`MediaPlayer.framework`) provides high-level support for playing audio and video content from your app. You can use this framework to do the following:

- Play video to a user's screen or to another device over AirPlay. You can play this video full screen or in a resizable view.
- Access the user's iTunes music library. You can play music tracks and playlists, search for songs, and present a media picker interface to the user.
- Configure and manage movie playback.
- Display Now Playing information in the lock screen and App Switcher. You can also display this information on an Apple TV when content is delivered via AirPlay.
- Detect when video is being streamed over AirPlay.

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

Metal Framework

Metal provides extremely low-overhead access to the A7 GPU enabling incredibly high performance for your sophisticated graphics rendering and computational tasks. Metal eliminates many performance bottlenecks—such as costly state validation—that are found in traditional graphics APIs. Metal is explicitly designed to move all expensive state translation and compilation operations out of the critical path of your most performance sensitive rendering code. Metal provides precompiled shaders, state objects, and explicit command scheduling to ensure your application achieves the highest possible performance and efficiency for your GPU graphics and compute tasks. This design philosophy extends to the tools used to build your app. When your app is built, Xcode compiles Metal shaders in the project into a default library, eliminating most of the runtime cost of preparing those shaders.

Graphics, compute, and blit commands are designed to be used together seamlessly and efficiently. Metal is specifically designed to exploit modern architectural considerations, such as multiprocessing and shared memory, to make it easy to parallelize the creation of GPU commands.

With Metal, you have a streamlined API, a unified graphics and compute shading language, and Xcode-based tools, so you don't need to learn multiple frameworks, languages and tools to take full advantage of the GPU in your game or app.

For more information about how to use Metal, see *Metal Programming Guide*, *Metal Framework Reference*, and *Metal Shading Language Guide*.

OpenAL Framework

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

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

OpenGL ES Framework

The OpenGL ES framework (`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 fine-grained graphics control and high frame rates for full-screen immersive apps such as games. You use the OpenGL framework in conjunction with the EAGL interfaces, which provide the interface between your OpenGL ES drawing calls and the native window objects in UIKit.

The framework supports OpenGL ES 1.1, 2.0, and 3.0. The 2.0 specification added support for fragment and vertex shaders and the 3.0 specification added support for many more features, including multiple render targets and transform feedback.

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

Photos Framework

The Photos framework (`Photos.framework`) provides new APIs for working with photo and video assets, including iCloud Photos assets, that are managed by the Photos app. This framework is a more capable alternative to the Assets Library framework. Key features include a thread-safe architecture for fetching and caching thumbnails and full-sized assets, requesting changes to assets, observing changes made by other apps, and resumable editing of asset content.

For more information about the interfaces of this framework, see *Photos Framework Reference*.

Photos UI Framework

The Photos UI framework (`PhotosUI.framework`) lets you create app extensions for editing image and video assets in the Photos app. For more information about how to create photo editing extensions, see *App Extension Programming Guide*.

Quartz Core Framework

The Quartz Core framework (`QuartzCore.framework`) contains the Core Animation interfaces. Core Animation is an advanced compositing technology that makes it easy to create view-based animations that are fast and efficient. The compositing engine takes advantage of the underlying hardware to manipulate your view's contents efficiently and in real time. Specify the start and end points of the animation, and let Core Animation do the rest. And because Core Animation is built in to the underlying `UIView` architecture, it is always available.

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

SceneKit Framework

SceneKit is an Objective-C framework for building simple games and rich app user interfaces with 3D graphics, combining a high-performance rendering engine with a high-level, descriptive API. SceneKit has been available since OS X v10.8 and is now available in iOS for the first time. Lower-level APIs (such as OpenGL ES) require you to implement the rendering algorithms that display a scene in precise detail. By contrast, SceneKit lets you describe your scene in terms of its content—geometry, materials, lights, and cameras—then animate it by describing changes to those objects.

SceneKit's 3D physics engine enlivens your app or game by simulating gravity, forces, rigid body collisions, and joints. Add high-level behaviors that make it easy to use wheeled vehicles such as cars in a scene, and add physics fields that apply radial gravity, electromagnetism, or turbulence to objects within an area of effect.

Use OpenGL ES to render additional content into a scene, or provide GLSL shaders that replace or augment SceneKit's rendering. You can also add shader-based post-processing techniques to SceneKit's rendering, such as color grading or screen space ambient occlusion.

For more information about the interfaces of this framework, see *SceneKit Framework Reference*.

SpriteKit Framework

The SpriteKit framework (`SpriteKit.framework`) provides a hardware-accelerated animation system for 2D and 2.5D games. SpriteKit provides the infrastructure that most games need, including a graphics rendering and animation system, sound playback support, and a physics simulation engine. Using SpriteKit frees you from creating these things yourself and lets you focus on the design of your content and the high-level interactions for that content.

Content in a SpriteKit app is organized into scenes. A scene can include textured objects, video, path-based shapes, Core Image filters, and other special effects. SpriteKit takes those objects and determines the most efficient way to render them onscreen. When it comes time to animate the content in your scenes, you can use SpriteKit to specify explicit actions you want to perform or use the physics simulation engine to define physical behaviors (such as gravity, attraction, or repulsion) for your objects.

In addition to the SpriteKit framework, there are Xcode tools for creating particle emitter effects and texture atlases. You can use the Xcode tools to manage app assets and update SpriteKit scenes quickly.

For more information about how to use SpriteKit, see *SpriteKit Programming Guide*. For an example of how to use SpriteKit to build a working app, see *code:Explained Adventure*.

Core Services Layer

The Core Services layer contains fundamental system services for apps. Key among these services are the Core Foundation and Foundation frameworks, which define the basic types that all apps use. This layer also contains individual technologies to support features such as location, iCloud, social media, and networking.

High-Level Features

The following sections describe some of the high-level features available in the Core Services layer.

Peer-to-Peer Services

The Multipeer Connectivity framework provides peer-to-peer connectivity over Bluetooth. You can use peer-to-peer connectivity to initiate communication sessions with nearby devices. Although peer-to-peer connectivity is used primarily in games, you can also use this feature in other types of apps.

For information about how to use peer-to-peer connectivity features in your app, see *Multipeer Connectivity Framework Reference* .

iCloud Storage

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

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

- **iCloud document storage.** Use this feature to store user documents and data in the user's iCloud account.
- **iCloud key-value data storage.** Use this feature to share small amounts of data among instances of your app.
- **CloudKit storage**. Use this feature when you want to create publicly shared content or when you want to manage the transfer of data yourself.

Most apps use iCloud document storage to share documents from a user's iCloud account. (This is the feature that users think of when they think of iCloud storage.) Users care about whether documents are shared across devices and whether they can see and manage those documents from a given device. In contrast, the iCloud key-value data store is not something users would see. Instead, it is a way for your app to share very small amounts of data (tens of kilobytes) with other instances of itself. Apps should use this feature to store noncritical app data, such as preferences, rather than important app data.

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

Block Objects

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

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

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

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

Data Protection

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

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

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

File-Sharing Support

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

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

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

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

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

Grand Central Dispatch

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

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

In-App Purchase

In-App Purchase gives you the ability to vend app-specific content and services and iTunes content from inside your app. This feature is implemented using the StoreKit framework, which provides the infrastructure needed to process financial transactions using the user's iTunes account. Your app handles the overall user experience and the presentation of the content or services available for purchase. For downloadable content, you can host the content yourself or let Apple's servers host it for you.

For more information about supporting in-app purchase, see *In-App Purchase Programming Guide* . For additional information about the StoreKit framework, see Store Kit Framework (page 46).

SQLite

The SQLite library lets you embed a lightweight SQL database into your app without running a separate remote database server process. From your app, 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, see the SQLite Software Library.

XML Support

The Foundation framework provides the `NSXMLParser` class for retrieving elements from an XML document. Additional support for manipulating XML content is provided by the `libxml2` library. This open source library lets you parse or write arbitrary XML data quickly and transform XML content to HTML.

The header files for accessing the `libxml2` library are located in the <*iOS_SDK*>`/usr/include/libxml2/` directory, where <*iOS_SDK*> is the path to the target SDK in your Xcode installation directory. For more information about using `libxml2`, see the documentation for libxml2.

Core Services Frameworks

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

Accounts Framework

The Accounts framework (`Accounts.framework`) provides a single sign-on model for certain user accounts. Single sign-on improves the user experience by eliminating the need to prompt the user separately for multiple accounts. It also simplifies the development model for you by managing the account authorization process for your app. You use this framework in conjunction with the Social framework.

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

Address Book Framework

The Address Book framework (`AddressBook.framework`) provides programmatic access to a user's contacts database. If your app uses contact information, you can use this framework to access and modify that information. 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.

> **Important:** Access to a user's contacts data requires explicit permission from the user. Apps must therefore be prepared for the user to deny that access. Apps are also encouraged to provide `Info.plist` keys describing the reason for needing access.

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

Ad Support Framework

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

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

CFNetwork Framework

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

- Use BSD sockets
- Create encrypted connections using SSL or TLS
- Resolve DNS hosts

- Work with HTTP servers, authenticating HTTP servers, and HTTPS servers
- Work with FTP servers
- Publish, resolve, and browse Bonjour services

CFNetwork is based, both physically and theoretically, on BSD sockets. For information on how to use CFNetwork, see *CFNetwork Programming Guide* and *CFNetwork Framework Reference*.

CloudKit Framework

CloudKit (`CloudKit.framework`) provides a conduit for moving data between your app and iCloud. Unlike other iCloud technologies where data transfers happen transparently, CloudKit gives you control over when transfers occur. You can use CloudKit to manage all types of data.

Apps that use CloudKit directly can store data in a repository that is shared by all users. This public repository is tied to the app itself and is available even on devices without a registered iCloud account. As the app developer, you can manage the data in this container directly and see any changes made by users through the CloudKit dashboard.

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

Core Data Framework

The Core Data framework (`CoreData.framework`) is a technology for managing the data model of a Model-View-Controller app. Core Data is intended for use in apps in which the data model is already highly structured. Instead of defining data structures programmatically, you use the graphical tools in Xcode to build a schema representing your data model. At runtime, instances of your data-model entities are created, managed, and made available through the Core Data framework.

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

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

If you are starting to develop a new app or are planning a significant update to an existing app, you should consider using Core Data. For an example of how to use Core Data in an iOS app, 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 apps. This framework includes support for the following:

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

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

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

Core Location Framework

The Core Location framework (`CoreLocation.framework`) provides location and heading information to apps. For location information, the framework uses the onboard GPS, cell, or Wi-Fi radios to find the user's current longitude and latitude. You can incorporate this technology into your own apps 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. Core Location also provides the following capabilities:

- Access to compass-based heading information on iOS devices that include a magnetometer

- Support for region monitoring based on a geographic location or Bluetooth beacon
- Support for low-power location-monitoring using cell towers
- Collaboration with MapKit to improve the quality of location data in specific situations, such as when driving

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

Core Media Framework

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

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

Core Motion Framework

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

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

Core Telephony Framework

The Core Telephony framework (`CoreTelephony.framework`) provides interfaces for interacting with phone-based information on devices that have a cellular radio. Apps can use this framework to get information about a user's cellular service provider. Apps interested in cellular call events (such as VoIP apps) 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*.

EventKit Framework

The EventKit framework (`EventKit.framework`) provides an interface for accessing calendar events on a user's device. You can use this framework to do the following:

- Get existing events and reminders from the user's calendar
- Add events to the user's calendar
- Create reminders for the user and have them appear in the Reminders app
- Configure alarms for calendar events, including setting rules for when those alarms should be triggered

> **Important:** Access to the user's calendar and reminder data requires explicit permission from the user. Apps must therefore be prepared for the user to deny that access. Apps are also encouraged to provide `Info.plist` keys describing the reason for needing access.

For more information about the classes and methods of this framework, see *EventKit Framework Reference*. See also Event Kit UI Framework (page 18).

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 41). 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*.

HealthKit Framework

HealthKit (`HealthKit.framework`) is a new framework for managing a user's health-related information. With the proliferation of apps and devices for tracking health and fitness information, it's difficult for users to get a clear picture of how they are doing. HealthKit makes it easy for apps to share health-related information, whether that information comes from devices connected to an iOS device or is entered manually by the user. The user's health information is stored in a centralized and secure location. The user can then see all of that data displayed in the Health app.

When your app implements support for HealthKit, it gets access to health-related information for the user and can provide information about the user, without needing to implement support for specific fitness-tracking devices. The user decides which data should be shared with your app. Once data is shared with your app, your app can register to be notified when that data changes; you have fine-grained control over when your app is notified. For example, you could request that your app be notified whenever the user takes his or her blood pressure, or be notified only when a measurement shows that the user's blood pressure reaches a specific reading.

For more information about the interfaces of this framework, see *HealthKit Framework Reference*.

HomeKit Framework

HomeKit (`HomeKit.framework`) is a new framework for communicating with and controlling connected devices in a user's home. New devices being introduced for the home are offering more connectivity and a better user experience. HomeKit provides a standardized way to communicate with those devices.

Your app can use HomeKit to communicate with devices that users have in their homes. Using your app, users can discover devices in their home and configure them. They can also create actions to control those devices. The user can group actions together and trigger them using Siri. Once a configuration is created, users can invite other people to share access to it. For example, a user might temporarily offer access to a house guest.

Use the HomeKit Accessory Simulator to test the communication of your HomeKit app with a device.

For more information about the interfaces of this framework, see *HomeKit Framework Reference*.

JavaScript Core Framework

The JavaScript Core framework (`JavaScriptCore.framework`) provides Objective-C wrapper classes for many standard JavaScript objects. Use this framework to evaluate JavaScript code and to parse JSON data.

For information about the classes of this framework, see the header files.

Mobile Core Services Framework

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*.

Multipeer Connectivity Framework

The Multipeer Connectivity framework (`MultipeerConnectivity.framework`) supports the discovery of nearby devices and the direct communication with those devices without requiring Internet connectivity. This framework makes it possible to create multipeer sessions easily and to support reliable in-order data transmission and real-time data transmission. With this framework, your app can communicate with nearby devices and seamlessly exchange data.

The framework provides programmatic and UI-based options for discovering and managing network services. Apps can integrate the `MCBrowserViewController` class into their UI to display a list of peer devices for the user to choose from. Alternatively, you can use the `MCNearbyServiceBrowser` class to look for and manage peer devices programmatically.

For more information about the interfaces of this framework, see *Multipeer Connectivity Framework Reference*.

NewsstandKit Framework

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

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

PassKit Framework

The Passbook app provides users with a place to store coupons, boarding passes, event tickets, and discount cards for businesses. Instead of carrying a physical representation of these items, users can now store them on their iOS device and use them the same way as before. The PassKit framework (`PassKit.framework`) provides the Objective-C interfaces you need to integrate support for these items into your apps. You use this framework in combination with web interfaces and file format information to create and manage the passes your company offers.

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

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

Quick Look Framework

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

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

Safari Services Framework

The Safari Services framework (`SafariServices.framework`) provides support for programmatically adding URLs to the user's Safari reading list. For information about the class provided by this framework, see the framework header files.

Social Framework

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

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

StoreKit Framework

The StoreKit framework (`StoreKit.framework`) provides support for the purchasing of content and services from within your iOS apps, a feature known as *In-App Purchase*. For example, you can use this feature to allow the user to unlock additional app features. Or if you are a game developer, you can use it to offer additional

game levels. In both cases, the StoreKit framework handles the financial aspects of the transaction, processing payment requests through the user's iTunes Store account and providing your app with information about the purchase.

StoreKit focuses on the financial aspects of a transaction, ensuring that transactions occur securely and correctly. Your app 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 app.

For information about how to use the StoreKit framework, see *In-App Purchase Programming Guide* and *StoreKit Framework Reference*.

System Configuration Framework

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

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

WebKit Framework

The WebKit framework (`WebKit.framework`) lets you display HTML content in your app. In addition to displaying HTML, you can provide basic editing support so that users can replace text and manipulate document text and attributes, including CSS properties. WebKit also supports creating and editing content at the DOM level of an HTML document. For example, you could extract the list of links on a page, modify them, and replace them prior to displaying the document in a web view.

For information about the interfaces of this framework, see *WebKit Framework Reference*.

Core OS Layer

The Core OS layer contains the low-level features that most other technologies are built upon. Even if you do not use these technologies directly in your apps, 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

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

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

Core Bluetooth Framework

The Core Bluetooth framework (`CoreBluetooth.framework`) allows developers to interact specifically with Bluetooth low energy (LE) accessories. The Objective-C interfaces of this framework allow you to do the following:

- Scan for Bluetooth accessories and connect and disconnect to ones you find
- Vend services from your app, turning the iOS device into a peripheral for other Bluetooth devices
- Broadcast iBeacon information from the iOS device
- Preserve the state of your Bluetooth connections and restore those connections when your app is subsequently launched
- Be notified of changes to the availability of Bluetooth peripherals

For more information about using the Core Bluetooth framework, see *Core Bluetooth Programming Guide* and *Core Bluetooth Framework Reference*.

External Accessory Framework

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

For more information about how to use this framework, see *External Accessory Programming Topics*. For information about developing accessories for iOS-based devices, go to the Apple Developer website.

Generic Security Services Framework

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

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

Local Authentication Framework

The Local Authentication Framework (`LocalAuthentication.framework`) lets you use Touch ID to authenticate the user. Some apps may need to secure access to all of their content, while others might need to secure certain pieces of information or options. In either case, you can require the user to authenticate before proceeding. Use this framework to display an alert to the user with an application-specified reason for why the user is authenticating. When your app gets a reply, it can react based on whether the user was able to successfully authenticate.

For more information about the interfaces of this framework, see *Local Authentication Framework Reference*.

Network Extension Framework

The Network Extension framework (NetworkExtension.framework) provides support for configuring and controlling Virtual Private Network (VPN) tunnels. Use this framework to create VPN configurations. You can then start VPN tunnels manually or supply on-demand rules to start the VPN tunnel in response to specific events.

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

Security Framework

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

The Common Crypto library provides additional support for symmetric encryption, hash-based message authentication codes (HMACs), and digests. The digests feature provides functions that are essentially compatible with those in the OpenSSL library, which is not available in iOS.

It is possible for you to share keychain items among multiple apps that you create. Sharing items makes it easier for apps in the same suite to interoperate 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 app separately. To share data between apps, you must configure the Xcode project of each app 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 Adding Capabilities in *App Distribution 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, based on Mach, is responsible for every aspect of the operating system. It manages the virtual memory system, threads, file system, network, and interprocess communication. The drivers at this layer also provide the interface between the available hardware and system frameworks. For security purposes, access to the kernel and drivers is restricted to a limited set of system frameworks and apps.

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

- Concurrency (POSIX threads and Grand Central Dispatch)
- 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*.

64-Bit Support

iOS was initially designed to support binary files on devices using a 32-bit architecture. In iOS 7, however, support was introduced for compiling, linking, and debugging binaries on a 64-bit architecture. All system libraries and frameworks are 64-bit ready, meaning that they can be used in both 32-bit and 64-bit apps. When compiled for the 64-bit runtime, apps may run faster because of the availability of extra processor resources in 64-bit mode.

iOS uses the LP64 model that is used by OS X and other 64-bit UNIX systems, which means fewer headaches when porting code. For information about the iOS 64-bit runtime and how to write 64-bit apps, see *64-Bit Transition Guide for Cocoa Touch*.

iOS Frameworks

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

Device Frameworks

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

Table A-1 Device frameworks

Name	First available	Prefixes	Description
`Accelerate.framework`	4.0	`cblas`, `vDSP`	Contains accelerated math and DSP functions. See Accelerate Framework (page 48).
`Accounts.framework`	5.0	`AC`	Contains interfaces for managing access to a user's system accounts. See Accounts Framework (page 39).
`AddressBook.framework`	2.0	`AB`	Contains functions for accessing the user's contacts database directly. See Address Book Framework (page 39).
`AddressBookUI.framework`	2.0	`AB`	Contains classes for displaying the system-defined people picker and editor interfaces. See Address Book UI Framework (page 18).
`AdSupport.framework`	6.0	`AS`	Contains a class for gathering analytics. See Ad Support Framework (page 39).

Name	First available	Prefixes	Description
AssetsLibrary.framework	4.0	AL	Contains classes for accessing the user's photos and videos. See Assets Library Framework (page 26).
AudioToolbox.framework	2.0	AU, Audio	Contains the interfaces for handling audio stream data and for playing and recording audio. See Core Audio (page 28).
AudioUnit.framework	2.0	AU, Audio	Contains the interfaces for loading and using audio units. See Core Audio (page 28).
AVFoundation.framework	2.2	AV	Contains Objective-C interfaces for playing and recording audio and video. See AV Foundation Framework (page 27).
AVKit.framework	8.0	AV	Contains Objective-C interfaces for playing and recording audio and video. See AVKit Framework (page 27).
CFNetwork.framework	2.0	CF	Contains interfaces for accessing the network via Wi-Fi and cellular radios. See CFNetwork Framework (page 39).
CloudKit.framework	8.0	CK	Contains Objective-C interfaces for fetching and saving iCloud data. See CloudKit (page 40).
CoreAudio.framework	2.0	Audio	Provides the data types used throughout Core Audio. See Core Audio (page 28).
CoreAudioKit.framework	8.0	CA	Provides the data types used throughout Core Audio. See CoreAudioKit Framework (page 28).
CoreBluetooth.framework	5.0	CB	Provides access to low-power Bluetooth hardware. See Core Bluetooth Framework (page 48).

Name	First available	Prefixes	Description
CoreData.framework	3.0	NS	Contains interfaces for managing your application's data model. See Core Data Framework (page 40).
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 (page 41).
CoreGraphics.framework	2.0	CG	Contains the interfaces for Quartz 2D. See Core Graphics Framework (page 29).
CoreImage.framework	5.0	CI	Contains interfaces for manipulating video and still images. See Core Image Framework (page 29).
CoreLocation.framework	2.0	CL	Contains the interfaces for determining a user's location. See Core Location Framework (page 41).
CoreMedia.framework	4.0	CM	Contains low-level routines for manipulating audio and video. See Core Media Framework (page 42).
CoreMIDI.framework	4.2	MIDI	Contains low-level routines for handling MIDI data. See Core Audio (page 28).
CoreMotion.framework	4.0	CM	Contains interfaces for accessing accelerometer and gyro data. See Core Motion Framework (page 42).
CoreTelephony.framework	4.0	CT	Contains routines for accessing telephony-related information. See Core Telephony Framework (page 42).
CoreText.framework	3.2	CT	Contains a text layout and rendering engine. See Core Text Framework (page 29).

Name	First available	Prefixes	Description
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 EventKit Framework (page 43).
EventKitUI.framework	4.0	EK	Contains classes for displaying the standard system calendar interfaces. See EventKit UI Framework (page 18).
External-Accessory.framework	3.0	EA	Contains interfaces for communicating with attached hardware accessories. See External Accessory Framework (page 49).
Foundation.framework	2.0	NS	Contains interfaces for managing strings, collections, and other low-level data types. See Foundation Framework (page 43).
GameController.framework	7.0	GC	Contains interfaces for communicating with game-related hardware. See Game Controller Framework (page 30).
GameKit.framework	3.0	GK	Contains interfaces for managing peer-to-peer connectivity. See GameKit Framework (page 18).
GLKit.framework	5.0	GLK	Contains Objective-C utility classes for building complex OpenGL ES applications. See GLKit Framework (page 30).
GSS.framework	5.0	gss	Provides a standard set of security-related services.
HealthKit.framework	8.0	HK	Provides a way to store health-related information for the user. See HealthKit Framework (page 44).

Name	First available	Prefixes	Description
HomeKit.framework	8.0	HM	Provides services for communicating with integrated household devices. See HomeKit Framework (page 44).
iAd.framework	4.0	AD	Contains classes for displaying advertisements in your application. See iAd Framework (page 19).
ImageIO.framework	4.0	CG	Contains classes for reading and writing image data. See Image I/O Framework (page 30).
IOKit.framework	2.0	N/A	Contains interfaces used by the device. Do not use this framework directly.
JavaScriptCore.framework	7.0	JS	Contains Objective-C wrappers for evaluating JavaScript code and parsing JSON. See JavaScript Core Framework (page 44).
Local-Authentication.framework	8.0	LA	Provides support for authenticating the user via Touch ID. See Local Authentication Framework (page 49).
MapKit.framework	3.0	MK	Contains classes for embedding a map interface into your application and for reverse-geocoding coordinates. See MapKit Framework (page 19).
Media-Accessibility.framework	7.0	MA	Manages the presentation of closed-caption content in media files. See Media Accessibility Framework (page 31).
MediaPlayer.framework	2.0	MP	Contains interfaces for playing full-screen video. See Media Player Framework (page 31).
MediaToolbox.framework	6.0	MT	Contains interfaces for playing audio content.
MessageUI.framework	3.0	MF	Contains interfaces for composing and queuing email messages. See Message UI Framework (page 19).

Name	First available	Prefixes	Description
Metal.framework	8.0	MTL	Provides a low-overhead graphics rendering engine for apps. See Metal Framework (page 31).
MobileCore-Services.framework	3.0	UT	Defines the uniform type identifiers (UTIs) supported by the system. See Mobile Core Services Framework (page 45).
Multipeer-Connectivity.framework	7.0	MC	Provides interfaces for implementing peer-to-peer networking between devices. See Multipeer Connectivity Framework (page 45).
Network-Extension.framework	8.0	NE	Provides interfaces for configuring and controlling VPN support. See Network Extension Framework (page 49).
NewsstandKit.framework	5.0	NK	Provides interfaces for downloading magazine and newspaper content in the background. See NewsstandKit Framework (page 45).
Notification-Center.framework	8.0	NK	Provides interfaces for implementing notification center widgets. See Notification Center Framework (page 20).
OpenAL.framework	2.0	AL	Contains the interfaces for OpenAL, a cross-platform positional audio library. See OpenAL Framework (page 32).
OpenGLES.framework	2.0	EAGL, GL	Contains the interfaces for OpenGL ES, which is an embedded version of the OpenGL cross-platform 2D and 3D graphics rendering library. See OpenGL ES Framework (page 32).
PassKit.framework	6.0	PK	Contains interfaces for creating digital passes to replace things like tickets, boarding passes, member cards, and more. See PassKit Framework (page 45).

Name	First available	Prefixes	Description
Photos.framework	8.0	PH	Contains interfaces for accessing and manipulating photo and videos. See Photos Framework (page 32).
PhotosUI.framework	8.0	PH	Contains interfaces for creating app extensions that manipulate photo and video assets. See Photos UI Framework (page 33).
PushKit.framework	8.0	PK	Provides a way for VoIP apps to register with a device. See PushKit Framework (page 20).
QuartzCore.framework	2.0	CA	Contains the Core Animation interfaces. See Quartz Core Framework (page 33).
QuickLook.framework	4.0	QL	Contains interfaces for previewing files. See Quick Look Framework (page 46).
SafariServices.framework	7.0	SS	Supports the creation of reading list items in Safari. See Safari Services Framework (page 46).
SceneKit.framework	8.0	SCN	Provides interfaces for creating 3D graphics. See SceneKit Framework (page 33).
Security.framework	2.0	CSSM, Sec	Contains interfaces for managing certificates, public and private keys, and trust policies. See Security Framework (page 50).
Social.framework	6.0	SL	Contains interfaces for interacting with social media accounts. See Social Framework (page 46).
SpriteKit.framework	7.0	SK	Facilitates the creation of sprite-based animations and rendering. See SpriteKit Framework (page 34).

Name	First available	Prefixes	Description
StoreKit.framework	3.0	SK	Contains interfaces for handling the financial transactions associated with in-app purchases. See StoreKit Framework (page 46).
System-Configuration.framework	2.0	SC	Contains interfaces for determining the network configuration of a device. See System Configuration Framework (page 47).
Twitter.framework	5.0	TW	Contains interfaces for sending tweets via the Twitter service. See Twitter Framework (page 20).
UIKit.framework	2.0	UI	Contains classes and methods for the iOS application user interface layer. See UIKit Framework (page 20).
VideoToolbox.framework	6.0	N/A	Contains interfaces used by the device. Do not include this framework directly.
WebKit.framework	8.0	WK	Provides support for integrating web content into your apps. See WebKit Framework (page 47).

Simulator Frameworks

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

For a list of the specific behavior differences between devices and iOS Simulator, see *iOS Simulator User 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 iOS SDK includes a local copy of the dynamic shared libraries that are installed with the system. These copies are installed on your development system so that you can link to them from your Xcode projects. To see the list of libraries for a particular version of iOS, look in `<`*Xcode.app*`>/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/<`*iOS_SDK*`>/usr/lib`, where `<`*Xcode.app*`>` is the path to your Xcode app and `<`*iOS_SDK*`>` is the specific SDK version you are targeting. For example, the shared libraries for the iOS 8.0 SDK would be located in the `/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS8.0.sdk/usr/lib` directory, with the corresponding headers in `/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS8.0.sdk/usr/include`.

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

Document Revision History

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

Date	Notes
2014-09-17	Updated to include features introduced in iOS 8.
2013-09-18	Added technologies introduced in iOS 7.
2012-09-19	Contains information about new frameworks and technologies introduced in iOS 6.
2011-10-12	Added technologies introduced in iOS 5.
2010-11-15	Updated the document to reflect new features in iOS 4.1 and iOS 4.2.
2010-07-08	Changed the title from "iPhone OS Technology Overview."
2010-06-04	Updated to reflect features available in iOS 4.0.
2009-10-19	Added links to reference documentation in framework appendix.
2009-05-27	Updated for iOS 3.0.
2008-10-15	New document that introduces iOS and its technologies.

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