

Apps

Apps are by far the predominant type of software created for Mac, or for any platform. You use Cocoa to build new Mac apps. To learn more about the features and frameworks available in Cocoa, see [Cocoa Application Layer](#).

In general, there are three basic styles of Mac apps:

The single–window utility app. A single–window utility app helps users perform the primary task within one window. Although a single–window utility app might also open an additional window—such as a preferences window—the user remains focused on the main window. Calculator is an example of a single–window utility app.

The single–window “shoebox” app. The defining characteristic of a shoebox app is the way it gives users an app–specific view of their content. For example, iPhoto users don’t find or organize their photos in the Finder; instead, they manage their photo collections entirely within the app.

The multiwindow document–based app. A multiwindow document–based app, such as Pages, opens a new window for each document the user creates or views. This style of app does not need a main window (although it might open a preferences or other auxiliary window).

App Extensions

No matter what type of app you write, you use app extensions to extend the functionality and content of that app to other parts of the system, or even to other apps. Types of extensions include:

Today. Display information from your app, or perform a quick task in the Today view of Notification Center.

Share. Share information with others by posting information to a website or social service, or sending data out in some other way.

Action. Create a context allowing the user to manipulate or view items from your app inside another app.

Finder. Show the sync state information in Finder.

Swift

Swift is a new programming language for Cocoa and Cocoa Touch with a concise and expressive syntax. Swift incorporates research on programming language combined with decades of experience building Apple platforms. Code written in Swift co–exists with existing classes written in Objective–C, allowing for easy adoption.

Some of the key features of Swift are:

Closures unified with function pointers

First class functions

Tuples and multiple return values

Generics

Fast and concise iteration over a range or collection

Structs and Enums that support methods, extensions, and protocols

Functional programming patterns

Type safety and type inference with restricted access to direct pointers

Swift also supports the use of Playgrounds, an interactive environment for real time evaluation of code. Use playgrounds for designing a new algorithm, creating and verifying new tests, or learning about the language and APIs.

To learn more about Swift, see *The Swift Programming Language (Swift 2.1)*, or for a quick overview, see *Welcome to Swift*.

Objective–C

Objective–C is a C–based programming language with object–oriented extensions. It is a primary development language for Cocoa apps. Unlike C++ and some other object–oriented languages, Objective–C comes with its own dynamic runtime environment. This runtime environment makes it much easier to extend the behavior of code at runtime without having access to the original source.

Objective-C 2.0 supports the following features, among many others:

Blocks (which are described in Block Objects)

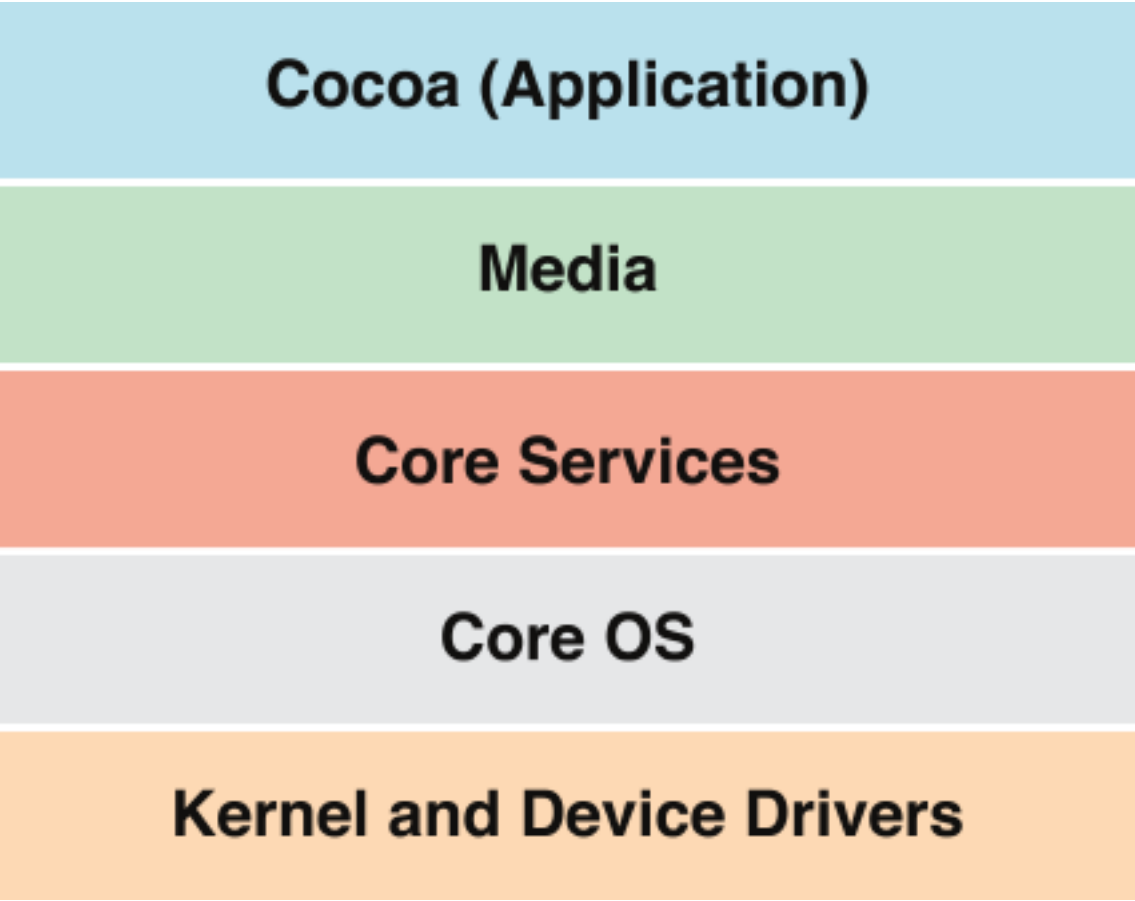
Declared properties, which offer a simple way to declare and implement an object’s accessor methods

A `for` operator syntax for performing fast enumerations of collections

Formal and informal protocols, which allow you to declare methods that are independent of any specific class, but which any class might implement

Categories and extensions, both of which allow you to add methods to an existing class

OS X Has a Layered Architecture with Key Technologies in Each Layer



The lower the layer a technology is in, the more specialized are the services it provides. Generally, technologies in higher layers incorporate lower-level technologies to provide common app behaviors. A good rule of thumb is to use the highest-level programming interface that meets the goals of your app.

The Cocoa (Application) layer includes technologies for building an app’s user interface, for responding to user events, and for managing app behavior.

The Cocoa application layer is primarily responsible for the appearance of apps and their responsiveness to user actions. In addition, many of the features that define the OS X user experience—such as Notification Center, full-screen mode, and Auto Save—are implemented by the Cocoa layer.

The Media layer encompasses specialized technologies for playing, recording, and editing audiovisual media and for rendering and animating 2D and 3D graphics.

The Core Services layer contains many fundamental services and technologies that range from Automatic Reference Counting and low-level network communication to string manipulation and data formatting.

The Core OS layer defines programming interfaces that are related to hardware and networking, including interfaces for running high-performance computation tasks on a computer’s CPU and GPU.

The Kernel and Device Drivers layer consists of the Mach kernel environment, device drivers, BSD library functions (`libSystem`), and other low-level components. The layer includes support for file systems, networking, security, interprocess communication, programming languages, device drivers, and extensions to the kernel

You Can Create Many Different Kinds of Software for Mac

Apps. Apps help users accomplish tasks that range from creating content and managing data to connecting with others and having fun. OS X provides a wealth of system technologies such as app extensions and handoff, that you use to extend the capabilities of your apps and enhance the experience of your users.

Frameworks and libraries. Frameworks and libraries enable code sharing among apps.

Command-line tools and daemons. Command-line tools allow sophisticated users to manipulate data in the command-line environment of the Terminal app. Daemons typically run continuously and act as servers for processing client requests.

App plug-ins and loadable bundles. Plug-ins extend the capabilities of other apps; bundles contain code and resources that apps can dynamically load at runtime.

System plug-ins. System plug-ins, such as audio units, kernel extensions, I/O Kit device drivers, preference panes, Spotlight importers, and screen savers, extend the capabilities of the system.

When Porting a Cocoa Touch App, Be Aware of API Similarities and Differences

The technology stacks on which Cocoa and Cocoa Touch apps are based have many similarities. Some system frameworks are identical (or nearly identical) in each platform, including Foundation, Core Data, and AV Foundation. This commonality of API makes some migration tasks—for example, porting the data model of your Cocoa Touch app—easy. (we are yet to discuss MVC)

Other migration tasks are more challenging because they depend on frameworks that reflect the differences between the platforms.

For example, porting controller objects and revising the user interface are more demanding tasks because they depend on AppKit and UIKit, which are the primary app frameworks in the Cocoa and CocoaTouch layers, respectively.

Cocoa Umbrella Framework

The Cocoa umbrella framework (`Cocoa.framework`) imports the core Objective-C frameworks for app development: AppKit, Foundation, and Core Data.

AppKit (`AppKit.framework`). This is the only framework of the three that is actually in the Cocoa layer. See [AppKit](#) for a summary of AppKit features and classes.

Foundation (`Foundation.framework`). The classes of the Foundation framework (which resides in the Core Services layer) implement data management, file access, process notification, network communication, and other low-level features. AppKit has a direct dependency on Foundation because many of its methods and functions either take instances of Foundation classes as parameters, or return the instances as values.

To find out more about Foundation, see [Foundation and Core Foundation](#).

Core Data (`CoreData.framework`). The classes of the Core Data framework (which also resides in the Core Services layer) manage the data model of an app based on the Model-View-Controller design pattern. Although Core Data is optional for app development, it is recommended for apps that deal with large data sets.

For more information about Core Data, see [Core Data](#).

AppKit

AppKit is the key framework for Cocoa apps. The classes in the AppKit framework implement the user interface (UI) of an app, including windows, dialogs, controls, menus, and event handling. They also handle much of the behavior required of a well-behaved app, including menu management, window management, document management, Open and Save dialogs, and pasteboard (Clipboard) behaviour.

In addition to having classes for windows, menus, event handling, and a wide variety of views and controls, AppKit has window- and data-controller classes and classes for fonts, colors, images, and graphics operations. A large subset of classes comprise the Cocoa text system, described in [Text, Typography, and Fonts](#). Other AppKit classes support document management, printing, and services such as spellchecking, help, speech, and pasteboard and drag-and-drop operations.

Apps can participate in many of the features that make the user experience of OS X such an intuitive, productive, and rewarding experience. These features include the following:

Gestures. Users appreciate being able to use fluid, intuitive Multi-Touch gestures to interact with OS X. AppKit classes make it easy to adopt these gestures in your app and to provide a better zoom experience without redrawing your content. For

example, `UIScrollView` includes built-in support for the smart zoom gesture (that is, a two-finger double-tap on a trackpad). When you provide the semantic layout of your content, `UIScrollView` can intelligently magnify the content under the pointer. You can also use this class to respond to the lookup gesture (that is, a three-finger tap on a trackpad). To learn more about the gesture support that `UIScrollView` provides, see [NSScrollView Class Reference](#).

Spaces. Spaces lets the user organize windows into groups and switch back and forth between groups to avoid cluttering up the desktop. AppKit provides support for sharing windows across spaces through the use of collection behavior attributes on the window. For information about setting these attributes, see [NSWindow Class Reference](#).

Fast User Switching. With this feature, multiple users can share access to a single computer without logging out. One user’s session can continue to run, while another user logs in and accesses the computer. To support fast user switching, be sure that your app avoids doing anything that might affect another version of the app running in a different session. To learn how to implement this behavior, see [Multiple User Environment Programming Topics](#).

Xcode includes Interface Builder, a user interface editor that contains a library of AppKit objects, such as controls, views, and controller objects. With it, you can create most of your UI (including much of its behavior) graphically rather than programmatically. With the addition of Cocoa bindings and Core Data, you can also implement most of the rest of your app graphically.

For an overview of the AppKit framework, see the introduction to the [AppKit Framework Reference](#). [Mac App Programming Guide](#) offers a practical discussion of how you use mostly AppKit classes to implement an app’s user interface, its documents, and its overall behaviour.

Core Graphics

The Quartz 2D client API offered by the Core Graphics framework (`CoreGraphics.framework`) provides commands for managing the graphics context and for drawing primitive shapes, images, text, and other content. The Core Graphics framework defines the Quartz 2D interfaces, types, and constants you use in your apps.

Quartz 2D provides many important features to apps, including the following:

High-quality rendering on the screen

High-resolution UI support

Antialiasing for all graphics and text

Support for adding transparency information to windows

Internal compression of data

A consistent feature set for all printers

Automatic PDF generation and support for printing, faxing, and saving as PDF

Color management through ColorSync

For information about the Quartz 2D API, see [Quartz 2D Programming Guide](#).

Video Technologies

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

When choosing a video technology, remember that the higher-level frameworks simplify your work and are, for this reason, usually preferred. The frameworks in the following list are ordered from highest to lowest level, with the AV Foundation framework offering the highest-level interfaces you can use.

AVKit (`AVKit.framework`). AV Kit supports playing visual content in your application using the standard controls.

AV Foundation (`AVFoundation.framework`). AV Foundation supports playing, recording, reading, encoding, writing, and editing audiovisual media.

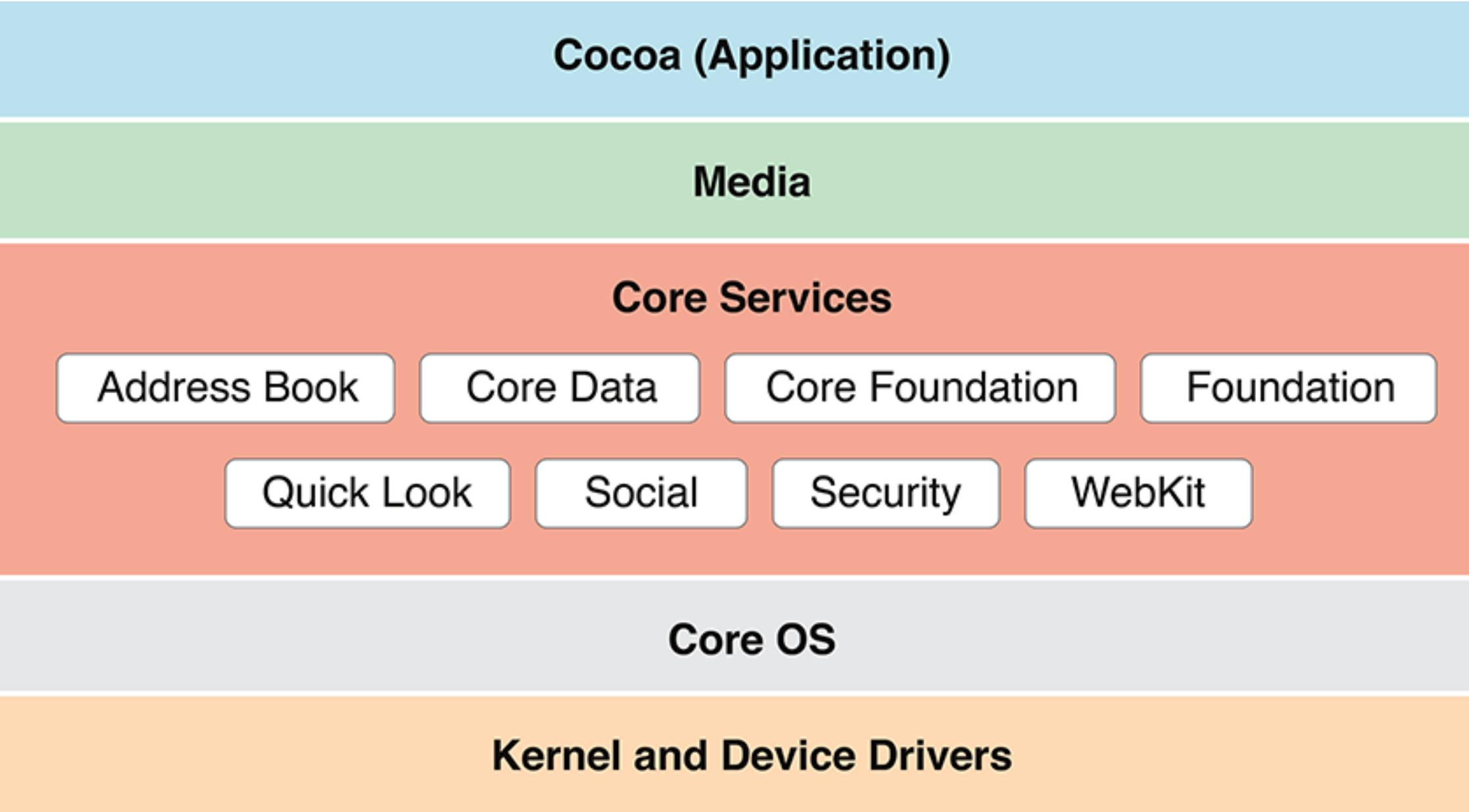
Core Media (`CoreMedia.framework`). Core Media provides a low-level C interface for managing audiovisual media. With the Core Media I/O framework, you can create plug-ins that can access media hardware and that can capture video and mixed audio and video streams.

Core Video (`CoreVideo.framework`). Core Video provides a pipeline model for digital video between a client and the GPU to deliver hardware-accelerated video processing while allowing access to individual frames. **Use Core Video only if your app needs to manipulate individual video frames; otherwise, use AV Foundation.**

For information about each of the video frameworks (as well as other frameworks) in the Media layer, see [Media Layer Frameworks](#).

Core Services Layer

The technologies in the Core Services layer are called *core services* because they provide essential services to apps but have no direct bearing on the app’s user interface. In general, these technologies are dependent on frameworks and technologies in the two lowest layers of OS X—that is, the Core OS layer and the Kernel and Device Drivers layer.



Core Service Frameworks

OS X includes several core services that make developing apps easier. These technologies range from utilities for managing your internal data structures to high-level frameworks for speech recognition and accessing calendar data. This section summarizes the technologies in the Core Services layer that are relevant to developers—that is, that have programmatic interfaces or have an impact on how you write software.

Core Services Umbrella Framework

The [Core Services umbrella framework](#) (`CoreServices.framework`) includes the following frameworks:

Launch Services (`LaunchServices.framework`). Launch Services gives you a programmatic way to open apps, documents, URLs, or files with a given MIME type in a way similar to the Finder or the Dock. The Launch Services framework also provides interfaces for programmatically registering the document types your app supports. Launch Services is in the Core Services umbrella framework. For information on how to use Launch Services, see *Launch Services Programming Guide*.

Metadata (`Metadata.framework`). The Metadata framework helps you to create Spotlight importer plug-ins. It also provides a query API that you can use in your app to search for files based on metadata values and then sort the results based on certain criteria. (The Foundation framework offers an Objective-C interface to the query API.) For more information on Spotlight importers, querying Spotlight, and the Metadata framework, see *Spotlight Importer Programming Guide* and *File Metadata Search Programming Guide*.

Search Kit (`SearchKit.framework`). Search Kit lets you search, summarize, and retrieve documents written in most human

languages. You can incorporate these capabilities into your app to support fast searching of content managed by your app. This framework is part of the Core Services umbrella framework. For detailed information about the available features, see [SearchKit Reference](#).

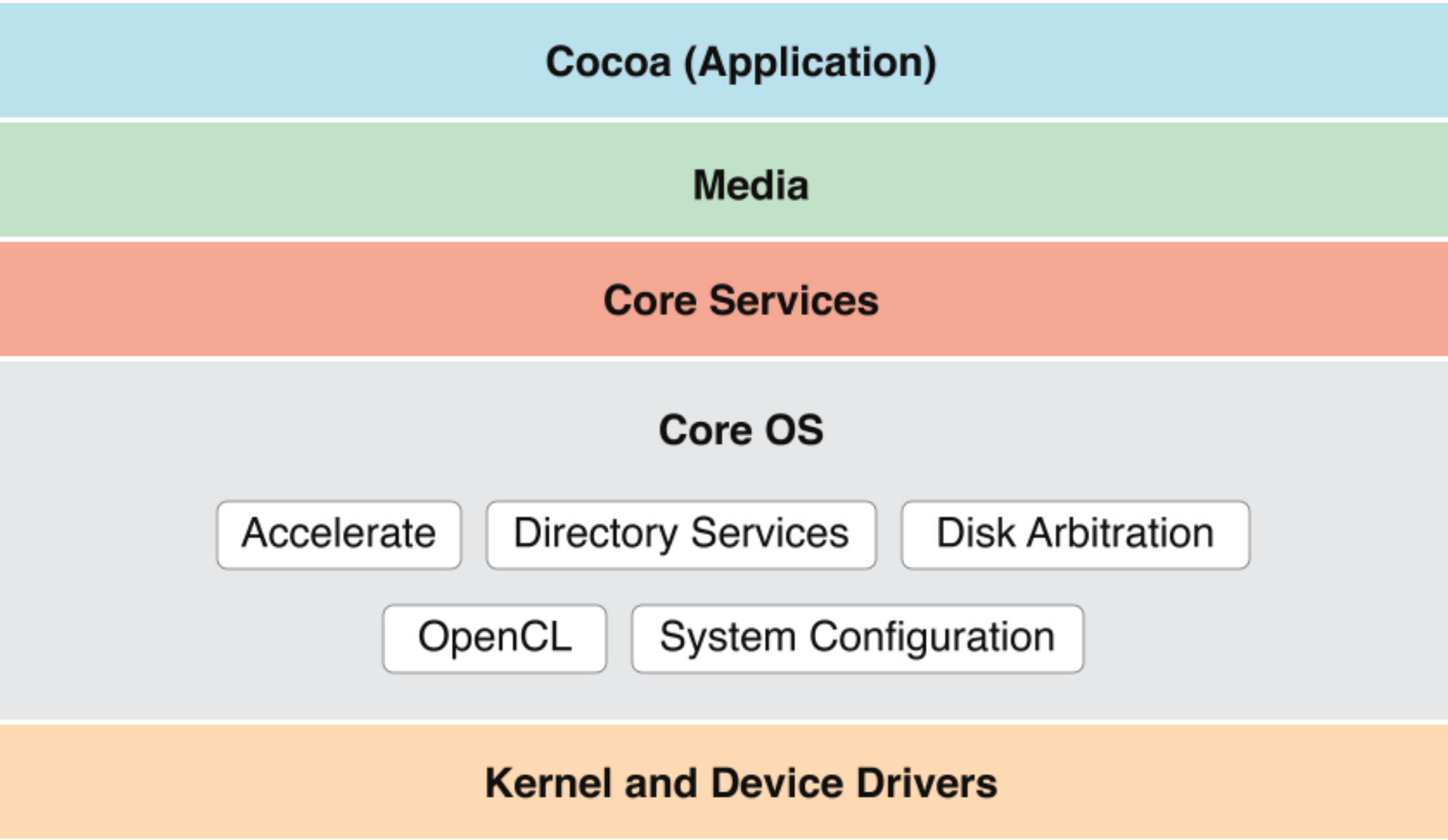
Web Services Core (`WebServicesCore.framework`). Web Services Core provides support for the invocation of web services using CFNetwork. The available services cover a wide range of information and include things such as financial data and movie listings. Web Services Core is part of the Core Services umbrella framework. For a description of web services and information on how to use the Web Services Core framework, see [Web Services Core Programming Guide](#).

Dictionary Services (`DictionaryServices.framework`). Dictionary Services lets you create custom dictionaries that users can access through the Dictionary app. Through these services, your app can also access dictionaries programmatically and can support user access to dictionary look-up through a contextual menu. For more information, see [Dictionary Services Programming Guide](#).

You should not link directly to any of these subframeworks; instead link to (and import) `CoreServices.framework`.

Core OS Layer

The technologies and frameworks in the Core OS layer provide low-level services related to hardware and networks. These services are based on facilities in the Kernel and Device Drivers layer.



High-Level Features

The Core OS layer implements features related to app security.

Gatekeeper

Gatekeeper, allows users to block the installation of software that does not come from the Mac App Store and identified developers. If your app is not signed with a Developer ID certificate issued by Apple, it will not launch on systems that have this security option selected. If you plan to distribute your app outside of the Mac App Store, be sure to test the installation of your app on a Gatekeeper enabled system so that you can provide a good user experience.

Xcode supports most of the tasks that you need to perform to get a Developer ID certificate and code sign your app. To learn how to submit your app to the Mac App Store—or test app installation on a Gatekeeper enabled system—read [Tools Workflow Guide for Mac](#).

App Sandbox

App Sandbox provides a last line of defense against stolen, corrupted, or deleted user data if malicious code exploits your app. App Sandbox also minimizes the damage from coding errors. Its strategy is twofold:

App Sandbox enables you to describe how your app interacts with the system. The system then grants your app only the access it needs to get its job done, and no more.

App Sandbox allows the user to transparently grant your app additional access by using Open and Save dialogs, drag and drop, and other familiar user interactions.

You describe your app’s interaction with the system by setting entitlements in Xcode. For details on all the entitlements available in OS X, see *Entitlement Key Reference*.

When you adopt App Sandbox, you must code sign your app (for more information, see [Code Signing](#)). This is because entitlements, including the special entitlement that enables App Sandbox, are built into an app’s code signature.

For a complete explanation of App Sandbox and how to use it, read *App Sandbox Design Guide*.

Core OS Frameworks

The following technologies and frameworks are in the Core OS layer of OS X:

Accelerate

The Accelerate framework (`Accelerate.framework`) contains APIs that help you accelerate complex operations—and potentially improve performance—by using the available vector unit. Hardware-based vector units boost the performance of any app that exploits data parallelism, such as those that perform 3D graphic imaging, image processing, video processing, audio compression, and software-based cell telephony. (Because Quartz and QuickTime Kit incorporate vector capabilities, any app that uses these APIs can tap into this hardware acceleration without making any changes.)

Disk Arbitration

The Disk Arbitration framework (`DiskArbitration.framework`) notifies your app when local and remote volumes are mounted and unmounted. It also furnishes other updates on the status of remote and local mounts and returns information about mounted volumes. For example, if you provide the framework with the BSD disk identifier of a volume, you can get the volume’s mount-point path.

For more information on Disk Arbitration, see *Disk Arbitration Framework Reference*.

OpenCL

The Open Computing Language (OpenCL) makes the high-performance parallel processing power of GPUs available for general-purpose computing. The OpenCL language is a general purpose computer language, not specifically a graphics language, that abstracts out the lower-level details needed to perform parallel data computation tasks on GPUs and CPUs. Using OpenCL, you create compute kernels that are then offloaded to a graphics card or CPU for processing. Multiple instances of a compute kernel can be run in parallel on one or more GPU or CPU cores, and you can link to your compute kernels from Cocoa, C, or C++ apps.

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.

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.

 iOS Developer Library

 Developer



iOS Technology Overview

PDF

Table of Contents

Introduction
Cocoa Touch Layer
Media Layer
Core Services Layer
Core OS Layer
Accelerate Framework
Core Bluetooth Framework
External Accessory Framework
Generic Security Services Framework
Local Authentication Framework
Network Extension Framework
Security Framework
System
64-Bit Support
Appendix A: iOS Frameworks
Revision History

[Previous](#)[Next](#)

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

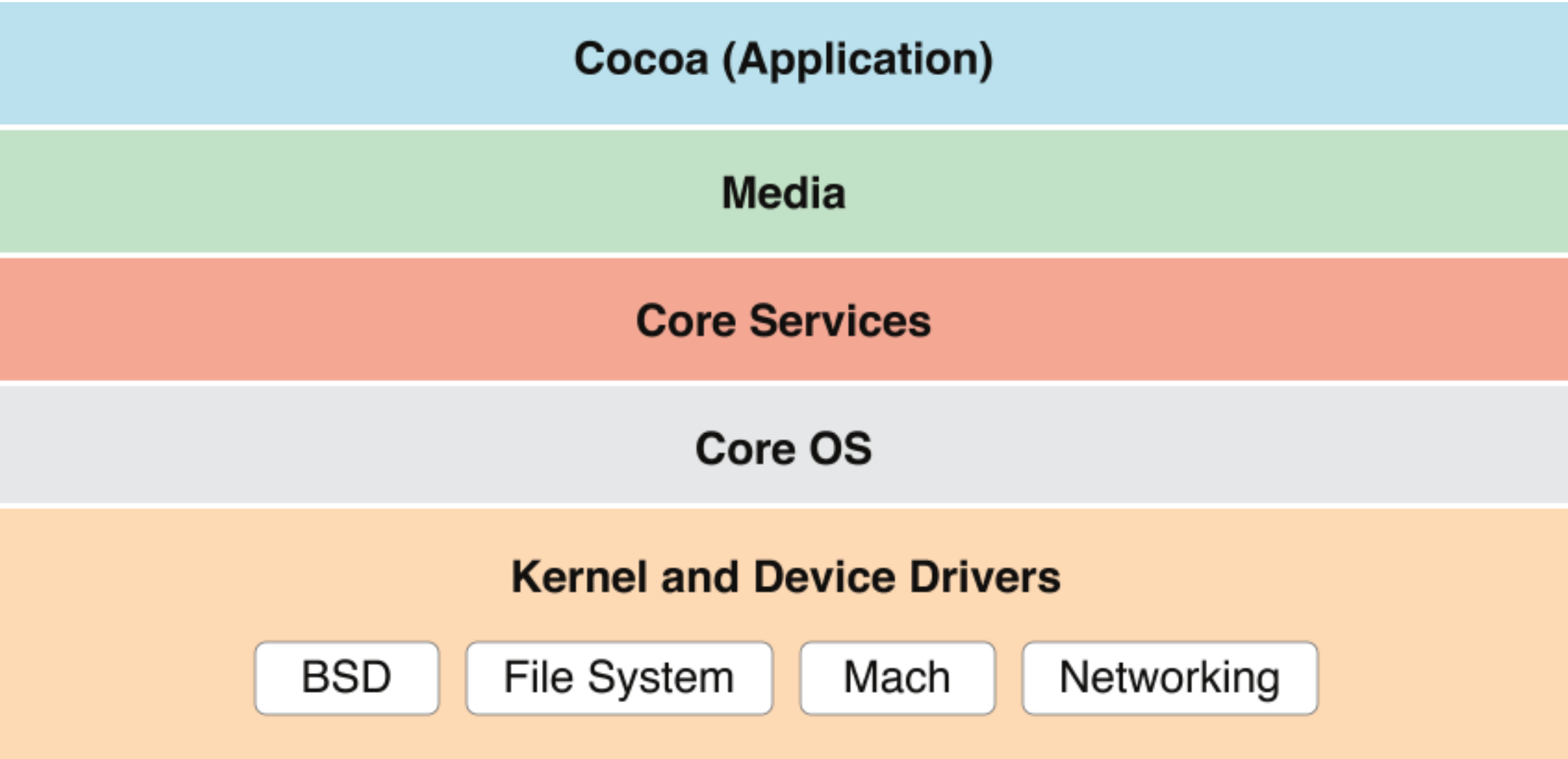
Memory allocation

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*.

Kernel and Device Drivers Layer

The lowest layer of OS X includes the kernel, drivers, and BSD portions of the system and is based primarily on open source technologies. OS X extends this low-level environment with several core infrastructure technologies that make it easier for you to develop software.



High-Level Features

The following sections describe features in the Kernel and Device Drivers layer of OS X.

XPC Interprocess Communication and Services

XPC is an OS X interprocess communication technology that complements App Sandbox by enabling privilege separation. Privilege separation, in turn, is a development strategy in which you divide an app into pieces according to the system resource access that each piece needs. The component pieces that you create are called *XPC services*.

Caching API

The `libcache` API is a low-level purgeable caching API. Aggressive caching is an important technique in maximizing app performance. However, when caching demands exceed available memory, the system must free up memory as necessary to handle new demands. Typically, this means paging cached data to and from relatively slow storage devices, sometimes even resulting in systemwide performance degradation. Your app should avoid potential paging overhead by actively managing its data caches, releasing them as soon as it no longer needs the cached data.

In-Kernel Video Capture

I/O Video provides a kernel-level C++ programming interface for writing video capture device drivers. I/O Video replaces the QuickTime sequence grabber API as a means of getting video into OS X.

I/O Video consists of the `IOVideoDevice` class on the kernel side (along with various related minor classes) that your driver should subclass, and a user space device interface for communicating with the driver.

For more information, see the `IOVideoDevice.h` header file in the Kernel framework.