Mac OS X Technology Overview

2006-06-28

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

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction to Mac OS X Technology Overview

Mac OS X is a modern operating system that combines a stable core with advanced technologies to help you deliver world-class products. The technologies in Mac OS X help you do everything from manage your data to display high-resolution graphics and multimedia content, all while delivering the consistency and ease of use that are hallmarks of the Macintosh experience.

This document offers a summary of the technologies available in Mac OS X and how you can use them to create outstanding products. Knowing what is available in the system can help you make good design decisions and deliver highly desirable products to your users.

Who Should Read This Document

Mac OS X Technology Overview is an essential guide for anyone who wants to develop software for Mac OS X. It orients all developers to the technologies available in Mac OS X, providing a quick reference with links to relevant documentation. This document also orients developers to the types of software they can create (besides applications), providing examples of when each of these types might be appropriate. Finally, the document offers guidelines and advice on the best ways to proceed with software development for Mac OS X.

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

Organization of This Document

This document has the following chapters and appendixes:

- "Mac OS X Architectural Overview" (page 15) provides background information for understanding the terminology and basic development environment of Mac OS X. It also provides a high-level overview of the Mac OS X system architecture.
- "Software Development Overview" (page 21) describes the types of software you can create for Mac OS X and offers recommendations and examples of when to create software of each type.
- "System-Level Technologies" (page 37) describes the system-level technologies in Mac OS X that are relevant to developers. It explains the purpose of each technology and points to additional resources for more details.

- "Application-Level Technologies" (page 65) describes the application-level technologies in Mac OS X that are relevant to developers. It explains the purpose of each technology and points to additional resources for more details.
- "Choosing Technologies to Match Your Design Goals" (page 75) provides guidance on the technologies you can use to support the high-level design goals set forth in *Apple Human Interface Guidelines*.
- "Porting Tips" (page 83) provides starter advice for developers who are porting applications from Mac OS 9, Windows, and UNIX platforms.
- "Command Line Primer" (page 89) provides an introduction to the command-line interface for developers who have never used it before.
- "Mac OS X Frameworks" (page 95) lists the frameworks in `/System/Library/Frameworks` along with a description of their contents. Use this list to learn about specific frameworks.
- "Mac OS X Developer Tools" (page 105) describes the tools and applications you use to create software for Mac OS X.

Getting the Xcode Tools

Apple provides a comprehensive suite of developer tools for creating Mac OS X software. The Xcode Tools include applications to help you design, create, debug, and optimize your software. This tools suite also includes header files, sample code, and documentation for Apple technologies. You can download the Xcode Tools from the members area of the Apple Developer Connection (ADC) website (http://connect.apple.com/). Registration is required but free.

For additional information about the tools available for working with Mac OS X and its technologies, see "Mac OS X Developer Tools" (page 105).

Reporting Bugs

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

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

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

See Also

This document does not provide in-depth information on any one technology. However, it does point to relevant documents in the ADC Reference Library. References of the form "<*title*> in <*category*> Documentation" refer to documents in specific sections of the reference library.

The following sections list additional sources of information about Mac OS X and its technologies.

Developer Documentation

The Xcode Tools CD provides the tools you need for development as well as examples and developer documentation for you to browse. When you install the Xcode Tools, the Installer application puts developer documentation into the following locations:

- **General documentation**. Most documentation is installed in the `/Developer/ADC Reference Library/documentation` directory. All documents are available in HTML format, which you can view from any web browser. You can also view the documentation from the Xcode IDE by choosing Help > Show Documentation Window.
- **Sample code**. A variety of sample programs are installed in `/Developer/Examples`, illustrating how to perform common tasks using the primary Mac OS X application environments: Carbon, Cocoa, and Java.

You can also get the latest documentation, release notes, Tech Notes, technical Q&As, and sample code from the ADC Reference Library (http://developer.apple.com/referencelibrary). All documents are available in HTML and many are also available in PDF format.

Information on BSD

Many developers who are new to Mac OS X are also new to BSD, an essential part of the operating system's kernel environment. BSD (for Berkeley Software Distribution) is based on UNIX. Several excellent books on BSD and UNIX are available in bookstores.

You can also use the World Wide Web as a resource for information on BSD. Several organizations maintain websites with manuals, FAQs, and other sources of information on the subject. For information about related projects, see:

- The OpenDarwin project (http://www.opendarwin.org)
- The FreeBSD project (http://www.freebsd.org)
- The NetBSD project (http://www.netbsd.org)
- The OpenBSD project (http://www.openbsd.org)

For more references, see the bibliography in *Kernel Programming Guide*.

Darwin and Open Source Development

Apple is the first major computer company to make open source development a key part of its ongoing operating system strategy. Apple has released the source code to virtually all of the components of Darwin to the developer community and continues to update the Darwin code base to include improvements as well as security updates, bug fixes, and other important changes.

Darwin consists of the Mac OS X kernel environment, BSD libraries, and BSD command environment. For more information about Darwin and what it contains, see "Darwin" (page 37). For detailed information about the kernel environment, see *Kernel Programming Guide*.

Information about how to join the Darwin open source effort is available at http://developer.apple.com/darwin/.

Other Information on the Web

Apple maintains several websites where developers can go for general and technical information about Mac OS X.

- The Apple software products site (http://www.apple.com/software) provides general information about Apple software.
- The Apple hardware products site (http://www.apple.com/hardware) provides general information about Apple hardware.
- The Apple product information site (http://www.apple.com/macosx) provides general information about Mac OS X.
- The ADC Reference Library (http://developer.apple.com/referencelibrary) features the same documentation that is installed with the developer tools. It also includes new and regularly updated documents as well as legacy documentation.
- The Apple Care Knowledge Base (http://www.apple.com/support/) contains technical articles, tutorials, FAQs, and other information.

Mac OS X Architectural Overview

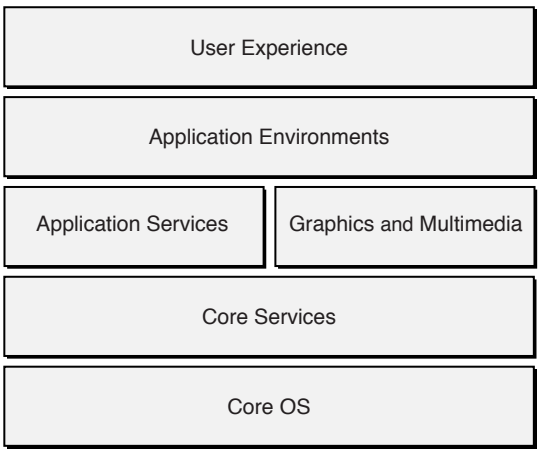
This chapter introduces some basic concepts you need to know before developing software for Mac OS X. The information presented here is high level; it is not a description of the inner workings of Mac OS X. Instead, the goal is to orient new developers to the basic architecture of Mac OS X and provide a platform for understanding the remaining information in this document.

Developers who are already familiar with the Mac OS X basic system architecture may want to skip this chapter and proceed directly to "Software Development Overview" (page 21). For information on specific terminology, see "Glossary" (page 131).

A Layered Look at the Mac OS X Architecture

A common way to look at complex software is to depict the software as a series of layers. Figure 1-1 shows one way of looking at the technologies of Mac OS X. The bottom layers of this diagram show fundamental building blocks of applications, including the kernel and low-level system services. Each new layer builds on the features in the previous layer. The result is the user experience provided by the applications and sophisticated technologies of Mac OS X.

Figure 1-1 Layers of Mac OS X



The Core OS layer of Mac OS X contains the kernel, device drivers, and low-level commands of the BSD environment. This is the heart of Mac OS X, providing a secure and stable foundation on which to build software. Most of the technologies in this layer are also part of Darwin. For details about the technologies in this layer, see "Core OS" (page 37).

The Core Services layer implements low-level features that can be used by most types of software. This layer includes features such as collection management, data formatting, memory management, string utilities, process management, XML parsing, stream-based I/O, and low-level network communication. Most of these features are included in the Core Foundation framework, which is described in more detail in "Core Foundation" (page 67).

The Application Services layer implements features that are geared more toward applications than other types of software. This layer includes features such as HTML rendering, disc recording, address book management, font management, speech synthesis and recognition, and many others. For more information about Application Services technologies, see "Application-Level Technologies" (page 65).

The Graphics and Multimedia layer implements specialized services for rendering 2D and 3D graphics, audio, and video. Quartz is the primary technology used for 2D rendering as well as for window management. OpenGL is an implementation of the industry-standard API for rendering 3D graphics. QuickTime is Apple's technology for displaying video, audio, virtual reality, and other multimedia-related information. Core Audio is Apple's technology for managing high-quality audio software and hardware. For more information about the available graphics and multimedia technologies and their architecture, see "Graphics, Imaging, and Multimedia" (page 46).

The Application Environments layer embodies the basic technologies for building applications. This is where most developers start their projects. Each application environment is tailored to the needs of its developers. For example, the Carbon environment is well suited for existing Mac OS developers or developers porting applications from other platforms; Cocoa is well suited for new developers or developers who want a modern object-oriented framework on which to build their applications. The Java and X11 environments are for developers who are already familiar with those environments and want their applications to run on multiple platforms, including Mac OS X. For more information about the available application environments, see "Application Environments" (page 21).

The User Experience layer is an abstraction that identifies methodologies for creating applications. Each methodology manifests itself as a set of guidelines, recommended technologies, or a combination of the two. For example, Aqua is a set of guidelines that describe the appearance of applications, whereas accessibility represents both a technology and a set of guidelines to support assistive technology devices such as screen readers. For more information about the technologies in the User Experience layer, see "User Experience" (page 58).

Mac OS X Runtime Architecture

The Mac OS X runtime architecture provides the underpinnings for launching and executing code on the platform. It is a good idea to understand this architecture before you start developing, because the environment you choose will affect the performance and behavior of your programs.

A runtime environment is a set of conventions that determines how code and data are loaded into memory and managed. When a program is launched, the runtime environment loads the program code into memory, resolves references to any external libraries, and prepares the code for execution.

Mac OS X supports two modern runtime environments: dyld (dynamic loader) and CFM (Code Fragment Manager). It also provides legacy support for Mac OS 9 applications through the Classic compatibility environment.

Important: With the transition to Intel-based processors, developers should create universal binaries for applications and build them for the dyld runtime environment and the Mach-O executable file format. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Dyld Runtime Environment

The dyld runtime environment is the preferred environment for any development you do because it is the native environment used by all Mac OS X system libraries. To support this environment, you must build your code modules using the Mach-O executable file format. The `dyld` library manager is the program responsible for loading your Mach-O code modules, resolving library dependencies, and beginning the execution of your code.

Upon loading your code, the `dyld` library manager performs the minimal amount of symbol binding needed to get your program up and running. This binding process involves loading whatever libraries contain the required symbols. After that, `dyld` binds symbols only as they are used by your code. If symbols are linked using weak linking, you must check to see if the symbol exists before using it.

CFM Runtime Environment

The CFM runtime environment is the legacy environment inherited from Mac OS 9. This environment supports applications that want to use modern features of Mac OS X but have not yet been converted over to the dyld environment for various reasons. Developers typically write programs for this environment so that the program can run in both Mac OS 9 and Mac OS X. The CFM runtime environment expects code modules to be built using the Preferred Executable Format (PEF).

Unlike the dyld environment, the CFM runtime environment takes a more static approach to symbol binding. At runtime, the CFM library manager binds all referenced symbols when the code modules are first loaded into memory. This binding occurs regardless of whether those symbols are actually used during the program's course of execution. If a particular symbol is missing, the program does not launch. (An exception to this rule occurs when code modules are bound together using weak linking, which explicitly permits symbols to be missing as long as they are never used.)

Because all system libraries are implemented using Mach-O and dyld, Mac OS X provides a set of libraries to bridge calls between CFM code and system libraries. This bridging is transparent but incurs a small amount of overhead for CFM-based programs. The Carbon library is one example of a bridged library.

Note: The libraries bridge only from CFM to dyld; they do not bridge calls going in the opposite direction. It is possible for a dyld-based application to make calls into a CFM-based library using the CFBundle facility, but this solution is not appropriate for all situations. If you want a library to be available to all Mac OS X execution environments, build it as a dyld-based library.

The Classic Environment

The Classic compatibility environment (or simply, Classic environment) makes it possible for Mac OS 9 applications to run unmodified in a Mac OS X system. Active development for the Classic environment is discouraged; however, developers with existing Mac OS 9 programs can read this section to understand the environment and its integration with the rest of Mac OS X.

Overview of the Classic Environment

The Classic environment is called a "software compatibility" environment because it enables Mac OS X to run applications built for Mac OS 9.1 or 9.2. The Classic environment is not intended to be a development environment. If you want to write programs to run in Mac OS X, you should always do so using the supported application environments described in "Application Environments" (page 21).

The Classic environment is not an emulator; it is a hardware abstraction layer between an installed Mac OS 9 System Folder and the Mac OS X kernel environment. Because of architectural differences, applications running in the Classic environment do not share the full advantages of the kernel environment. For example, they do not enjoy the benefits of memory protection and preemptive multitasking. Thus, if an application crashes or hangs in the Classic environment, it could take the whole environment down with it, forcing a restart. Luckily, it is only the Classic environment that has to be restarted, not the Mac OS X system that acts as its host.

Integration With Mac OS X

Wherever possible, Apple has tried to make the behavior of the Classic environment indistinguishable from that of Mac OS X. For example, copy/paste and drag-and-drop operations work between the two environments. However, not everything looks or behaves the same. For example, windows and menus still retain their familiar Mac OS 9 appearance.

The Classic environment supports all of the file systems supported by Mac OS X: Mac OS Standard (HFS), Mac OS Extended (HFS+), AFP, ISO 9660, UDF, NFS, UFS, SMB, WebDAV, and so on. All file sharing implementations in Classic go through Mac OS X.

The Classic environment disables any system extensions that conflict with features provided by Mac OS X. For example, Classic disables the Multiple Users extension because of the inherent support for multiple users in Mac OS X. Similarly, many preferences set using the Mac OS X System Preferences override similar preferences set using control panels in the Classic environment.

Networking in the Classic environment is largely integrated with the networking facilities of Mac OS X. Classic shares the Mac OS X networking devices (Ethernet, AirPort, and PPP), the computer's IP address, and the IP port address space. However, Classic runs the full Open Transport protocol stack (with full streams support), whereas the partial implementation of Open Transport in Carbon is built on top of BSD sockets.

If you set up or change printers in Mac OS X, the system tries to make those changes available to the Classic environment. Printer setup in the Classic environment uses the Chooser or Desktop Printer Utility, just as it does in a native Mac OS 9 system. When printing from an application in the Classic environment, users select a printer using a pop-up menu in the Print dialog.

Here are a few other areas of Classic integration that are of general interest:

- **Fonts**. The Apple Type Services (ATS) system integrates and manages fonts stored in the Fonts folder of the Classic System Folder. Thus fonts in that folder are shared throughout Mac OS X. However, fonts placed in other Font folders of Mac OS X are not shared with the Classic environment.
- **AppleScript**. AppleScript in the Classic environment is aware of application packages, which means you can communicate with Mac OS X applications from the Classic environment.
- **Processes**. Classic sees all Mac OS X applications in its process list, including many processes not shown in the Force Quit window in Mac OS X.
- **Copy/paste**. Copying data from a Cocoa application to the Classic environment might fail. Copying from Classic to Cocoa should work in most cases because Classic uses the Carbon data types for specifying data.

Software Development Overview

There are many ways to create an application in Mac OS X. There are also many types of software that you can create besides applications. The following sections introduce the types of software you can create in Mac OS X and when you might consider doing so.

Application Environments

Applications are by far the predominant type of software created for Mac OS X, or for any platform. Mac OS X provides numerous environments for developing applications, each of which is suited for specific types of development. The following sections describe each of the primary application environments and offer guidelines to help you choose an environment that is appropriate for your product requirements.

Important: With the transition to Intel-based processors, developers should always create universal binaries for their Carbon, Cocoa, and BSD applications. Java and WebObjects may also need to create universal binaries for bridged code. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Carbon

Based on the original Mac OS 9 interfaces, the Carbon application environment is a set of C APIs used to create full-featured applications for all types of users. The Carbon environment includes support for all of the standard Aqua user interface elements such as windows, controls, and menus, which you can design using Interface Builder. It also provides an extensive infrastructure for handling events, managing data, and using system resources.

The Carbon environment is especially suited for:

- Mac OS 9 developers porting their applications to Mac OS X
- Developers who prefer to work solely in C or C++
- Developers who are porting commercial applications from other procedural-based systems and want to use as much of their original code as possible

Because the Carbon interfaces are written in C, some developers may find them more familiar than the interfaces in the Cocoa or Java environments. Many C++ developers also prefer the Carbon environment for development, although C++ code can be integrated seamlessly into Cocoa applications as well.

The Carbon APIs offer you complete control over the features in your application; however, that control comes at the cost of added complexity. Whereas Cocoa provides many features for you automatically, with Carbon you must write the code to support those features yourself. For example, Cocoa applications automatically include default event handlers, pasteboard support, and Apple event support, but Carbon developers must add support for these features to their applications using the available Carbon APIs.

The Carbon application environment comprises several key umbrella frameworks, including the Carbon framework (`Carbon.framework`), the Core Services framework (`CoreServices.framework`), and the Application Services framework (`ApplicationServices.framework`). The Carbon environment also uses the Core Foundation framework (`CoreFoundation.framework`) extensively in its implementation.

Apple's developer documentation contains a section devoted to Carbon, where you can find conceptual material, reference documentation, and tutorials showing how to write applications using Carbon. See Getting Started With Carbon in Carbon Documentation for an overview of the available Carbon documentation.

If you are migrating a Mac OS 9 application to Mac OS X, read *Carbon Porting Guide*. If you are migrating from Windows, see *Porting to Mac OS X from Windows Win32 API*. If you are migrating from UNIX, see *Porting UNIX/Linux Applications to Mac OS X*.

Cocoa

The Cocoa application environment is designed for rapid application development. It features a sophisticated object-oriented framework and set of graphical tools that enable you to create full-featured applications quickly and without a lot of code. The Cocoa environment is especially suited for:

- New developers
- Developers who need to prototype an application quickly
- Developers who prefer to leverage the default behavior provided by the Cocoa frameworks so they can focus on the features unique to their application
- Objective-C or Objective-C++ developers

The objects in the Cocoa framework handle much of the behavior required of a well-behaved Mac OS X application, including menu management, window management, document management, Open and Save dialogs, and pasteboard (clipboard) behavior. Using Interface Builder, you create most of your application interface graphically rather than programatically. With the addition of Cocoa bindings, you can also implement much of your data display and layout graphically.

The Cocoa application environment consists of two object-oriented frameworks: Foundation (`Foundation.framework`) and the Application Kit (`AppKit.framework`). The classes in the Foundation framework implement data management, file access, process notification, memory management, network communication, and other low-level features. The classes in the Application Kit framework implement the user interface layer of an application, including windows, dialogs, controls, menus, and event handling. If you are writing an application, link with the Cocoa framework

(`Cocoa.framework`), which imports both Foundation and the Application Kit. If you are writing a Cocoa program that does not have a graphical user interface (a background server, for example), you can link your program solely with the Foundation framework.

Apple's developer documentation contains a section devoted to Cocoa where you can find conceptual material, reference documentation, and tutorials showing how to write Cocoa applications. For an introduction to the Cocoa environment, see *Cocoa Fundamentals Guide*. For information about the development tools, including Interface Builder, see "Mac OS X Developer Tools" (page 105). For information about Cocoa bindings, see *Cocoa Bindings Programming Topics*.

Java

The Java application environment is a runtime environment and set of objects for creating applications that run on multiple platforms. The Java environment is especially suited for:

- Experienced Java 2 Platform, Standard Edition (J2SE) developers
- Developers writing applications to run on multiple platforms
- Developers writing Java applets for inclusion in web-based content
- Developers familiar with the Swing or AWT toolkits for creating graphical interfaces

The Java application environment lets you develop and execute 100% pure Java applications and applets. This environment conforms to the specifications laid out by the J2SE platform, including those for the Java virtual machine (JVM), making applications created with this environment very portable. You can run them on computers with a different operating system and hardware as long as that system is running a compatible version of the JVM. Java applets should run in any Internet browser that supports them.

Note: If your Java application includes bridged code in a Mach-O binary, you need to create a universal binary for the Mach-O binary code. For more information, see *Universal Binary Programming Guidelines, Second Edition*.

For details on the tools and support provided for Java developers, see "Java Support" (page 45).

AppleScript

The AppleScript application environment lets you use AppleScript scripts to quickly create native Mac OS X applications that support the Aqua user interface guidelines. At the heart of this environment is AppleScript Studio, which combines features from AppleScript with Xcode, Interface Builder, and the Cocoa application framework. Using these tools, you can create applications that use AppleScript scripts to control a broad range of Cocoa user-interface objects.

AppleScript Studio has something to offer both to scripters and to those with Cocoa development experience. In addition to AppleScript's ability to control multiple applications, including parts of the Mac OS itself, you can use it for the following:

- Scripters can now create applications with windows, buttons, menus, text fields, tables, and much more. Scripts have full access to user interface objects.

- Cocoa developers can take advantage of AppleScript Studio's enhanced Cocoa scripting support, which can be useful in prototyping, testing, and deploying applications.

For information on how to create applications using AppleScript Studio, see *AppleScript Studio Programming Guide*.

WebObjects

The WebObjects application environment is a set of tools and object-oriented frameworks targeted at developers creating web services and web-based applications. The WebObjects environment provides a set of flexible tools for creating full-featured web applications. Common uses for this environment include the following:

- Creating a web-based interface for dynamic content, including programmatically generated content or content from a database
- Creating web services based on SOAP, XML, and WSDL

WebObjects is a separate product sold by Apple. If you are thinking about creating a web storefront or other web-based services, see the information available at http://developer.apple.com/tools/webobjects.

Note: If your WebObjects application includes bridged code in a Mach-O binary, you need to create a universal binary for the Mach-O binary code. For more information, see *Universal Binary Programming Guidelines, Second Edition*.

BSD and X11

The BSD application environment is a set of low-level interfaces for creating shell scripts, command-line tools, and daemons. The BSD environment is especially suited for:

- UNIX developers familiar with the FreeBSD and POSIX interfaces
- Developers who want to create text-based scripts and tools, rather than tools that have a graphical user interface
- Developers who want to provide fundamental system services through the use of daemons or other root processes

The BSD environment is for developers who need to work below the user interface layers provided by Carbon, Cocoa, and WebObjects. Developers can also use this environment to write command-line tools or scripts to perform specific user-level tasks.

X11 extends the BSD environment by adding a set of programming interfaces for creating graphical applications that can run on a variety of UNIX implementations. The X11 environment is especially suited for developers who want to create graphical applications that are also portable across different varieties of UNIX.

The BSD environment is part of the Darwin layer of Mac OS X. For information about Darwin, see Darwin Documentation. For more information about X11 development, see http://developer.apple.com/darwin/projects/X11. See also "Information on BSD" (page 13) for links to additional BSD resources.

Frameworks

A framework is a special type of bundle used to distribute shared resources, including library code, resource files, header files, and reference documentation. Frameworks offer a more flexible way to distribute shared code that you might otherwise put into a dynamic shared library. Whereas image files and localized strings for a dynamic shared library would normally be installed in a separate location from the library itself, in a framework they are integral to the framework bundle. Frameworks also have a version control mechanism that makes it possible to distribute multiple versions of a framework in the same framework bundle.

Apple uses frameworks to distribute the public interfaces of Mac OS X. You can use frameworks to distribute public code and interfaces created by your company. You can also use frameworks to develop private shared libraries that you can then embed in your applications.

Note: Mac OS X also supports the concept of "umbrella" frameworks, which encapsulate multiple subframeworks in a single package. However, this mechanism is used primarily for the distribution of Apple software. The creation of umbrella frameworks by third-party developers is not recommended.

You can develop frameworks using any programming language you choose; however, it is best to choose a language that makes it easy to update the framework later. Apple frameworks generally export programmatic interfaces in either ANSI C or Objective-C. Both of these languages have a well-defined export structure that makes it easy to maintain compatibility between different revisions of the framework. Although it is possible to use other languages when creating frameworks, you may run into binary compatibility problems later when you update your framework code.

For information on the structure and composition of frameworks, see *Framework Programming Guide*. That document also contains details on how to create public and private frameworks with Xcode.

Important: With the transition to Intel-based processors, developers should always create universal binaries for frameworks written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Plug-ins

Plug-ins are the standard way to extend many applications and system behaviors. Plug-ins are bundles whose code is loaded dynamically into the runtime of an application. Because they are loaded dynamically, they can be added and removed by the user.

There are many opportunities for developing plug-ins for Mac OS X. Developers can create plug-ins for third-party applications or for Mac OS X itself. Some parts of Mac OS X define plug-in interfaces for extending the basic system behavior. The following sections list many of these opportunities for developers, although other software types may also use the plug-in model.

Important: With the transition to Intel-based processors, developers should always create universal binaries for plug-ins written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Address Book Action Plug-Ins

An Address Book action plug-in lets you populate the pop-up menus of the Address Book application with custom menu items that use Address Book data to trigger a specific event. For example, you could add an action to a phone number field to trigger the dialing of the number using a Bluetooth-enabled phone.

Address Book action plug-ins are best suited for developers who want to extend the behavior of the Address Book application to support third-party hardware or software. For more information on creating an Address Book action plug-in, see the documentation for the ABActionDelegate class.

Application Plug-Ins

Several applications, including iTunes, iMovie, Final Cut Pro, and Final Cut Express, use plug-ins to extend the features available from the application. You can create plug-ins to implement new effects for these applications or for other applications that support a plug-in model. For information about developing plug-ins for Apple applications, visit the ADC website at http://developer.apple.com/.

Automator Plug-Ins

Introduced in Mac OS X version 10.4, Automator is a workflow-based application that lets users assemble complex scripts graphically using a palette of available actions. You can extend the default set of actions by creating Automator plug-ins to support new actions. Because they can be written in AppleScript or Objective-C, you can write plug-ins for your own application's features or for the features of other scriptable applications.

If you are developing an application, you should think about providing Automator plug-ins for your application's most common tasks. AppleScript is one of the easiest ways for you to create Automator plug-ins because it can take advantage of existing code in your application. If you are an Objective-C developer, you can also use that language to create plug-ins.

For information on how to write an Automator plug-in, see *Automator Programming Guide*.

Contextual Menu Plug-Ins

The Finder associates contextual menus with file-system items to give users a way to access frequently used commands quickly. Third-party developers can extend the list of commands found on these menus by defining their own contextual menu plug-ins. You might use this technique to make frequently used features available to users without requiring them to launch your application. For example, an archiving program might provide commands to compress a file or directory.

The process for creating a contextual menu plug-in is similar to that for creating a regular plug-in. You start by defining the code for registering and loading your plug-in, which might involve creating a factory object or explicitly specifying entry points. To implement the contextual menu behavior, you must then implement several callback functions defined by the Carbon Menu Manager for that purpose. Once complete, you install your plug-in in the `Library/Contextual Menu Items` directory at the appropriate level of the system, usually the local or user level.

For information on how to create a plug-in, see *Plug-ins*. For information on the Carbon Menu Manager functions you need to implement, see *Menu Manager Reference*.

Core Audio Plug-Ins

The Core Audio system supports plug-ins for manipulating audio streams during most processing stages. You can use plug-ins to generate, process, receive, or otherwise manipulate an audio stream. You can also create plug-ins to interact with new types of audio-related hardware devices.

For an introduction to the Core Audio environment, download the Core Audio SDK from http://developer.apple.com/sdk/ and read the documentation that comes with it. Information is also available in Music & Audio Core Audio Documentation.

Image Units

In Mac OS X version 10.4 and later, you can create **image units** for the Core Image and Core Video technologies. An image unit is a collection of filters packaged together in a single bundle. Each filter implements a specific manipulation for image data. For example, you could write a set of filters that perform different kinds of edge detection and package them as one image unit.

For more information about Core Image, see *Core Image Programming Guide*.

Input Method Components

An input method component is a code module that processes incoming data and returns an adjusted version of that data. A common example of an input method is an interface for typing Japanese or Chinese characters using multiple keystrokes. The input method processes the user keystrokes and returns the complex character that was intended. Other examples of input methods include spelling checkers and pen-based gesture recognition systems.

Input method components are implemented using the Carbon Component Manager. An input method component provides the connection between Mac OS X and any other programs your input method uses to process the input data. For example, you might use a background application to record the input keystrokes and compute the list of potential complex characters that those keystrokes can create.

For information on how to create a basic input method, see the *BasicInputMethod* sample code. For information on how to create components, see the *Component Manager Reference*.

Metadata Importers

In Mac OS X version 10.4 and later, you can create a metadata importer for your application's file formats. Metadata importers are used by Spotlight to gather information about the user's files and build a systemwide index. This index is then used for advanced searching based on more user-friendly information.

If your application defines a custom file format, you should always provide a metadata importer for that file format. If your application relies on commonly used file formats, such as JPEG, RTF, or PDF, the system provides a metadata importer for you.

For information on creating metadata importers, see *Spotlight Importer Programming Guide*.

QuickTime Components

A QuickTime component is a plug-in that provides services to QuickTime-savvy applications. The component architecture of QuickTime makes it possible to extend the support for new media formats, codecs, and hardware. Using this architecture, you can implement components for the following types of operations:

- Compressing/decompressing media data
- Importing/exporting media data
- Capturing media data
- Generating timing signals
- Controlling movie playback
- Implementing custom video effects, filters, and transitions
- Streaming custom media formats

For an overview of QuickTime components, see *QuickTime Overview*. For information on creating specific component types, see the subcategories in QuickTime Documentation.

Safari Plug-ins

Beginning with Mac OS X version 10.4, Safari supports a new plug-in model for tying in additional types of content to the web browser. This new model is based on an Objective-C interface and offers significant flexibility to plug-in developers. In particular, the new model lets plug-in developers take advantage of the Tiger API for modifying DOM objects in an HTML page. It also offers hooks so that JavaScript code can interact with the plug-in at runtime.

Safari plug-in support is implemented through the new WebPlugIn object and related objects defined in Web Kit. For information about how to use these objects, see *Web Kit Plug-In Programming Topics* and *Web Kit Objective-C Framework Reference*.

Dashboard Widgets

Introduced in Mac OS X version 10.4 and later, Dashboard provides a lightweight desktop layer for running **widgets**. Widgets are lightweight web applications that display information a user might use occasionally. You could write widgets to track stock quotes, view the current time, or access key features of a frequently used application. Widgets reside in the Dashboard layer, which is activated by the user and comes into the foreground in a manner similar to Exposé. Mac OS X comes with several standard widgets, including a calculator, clock, and iTunes controller.

Creating widgets is simpler than creating most applications because widgets are effectively HTML-based applications with optional JavaScript code to provide dynamic behavior. Dashboard uses the Web Kit to provide the environment for displaying the HTML and running the JavaScript code. Your widgets can take advantage of several extensions provided by that environment, including a way to render content using Quartz-like JavaScript functions.

For information on how to create widgets, see *Dashboard Tutorial*.

Agent Applications

An agent is a special type of application designed to help the user in an unobtrusive manner. Agents typically run in the background, providing information as needed to the user or to another application. Agents can display panels occasionally or come to the foreground to interact with the user if necessary. User interactions should always be brief and have a specific goal, such as setting preferences or requesting a piece of needed information.

An agent may be launched by the user but is more likely to be launched by the system or another application. As a result, agents do not show up in the Dock or the Force Quit window. Agents also do not have a menu bar for choosing commands. User manipulation of an agent typically occurs through dialogs or contextual menus in the agent user interface.

The iChat application uses an agent to communicate with the chat server and notify the user of incoming chat requests. The Dock is another agent program that is launched by the system for the benefit of the user.

Important: With the transition to Intel-based processors, developers should always create universal binaries for applications written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Screen Savers

Screen savers are small programs that take over the screen after a certain period of idle activity. Screen savers provide entertainment and also prevent the screen image from being burned into the surface of a screen permanently. Mac OS X supports both slideshows and programmatically generated screen-saver content.

Slideshows

A slideshow is a simple type of screen saver that does not require any code to implement. To create a slideshow, you create a bundle with an extension of `.slideSaver`. Inside this bundle, you place a Resources directory containing the images you want to display in your slideshow. Your bundle should also include an information property list that specifies basic information about the bundle, such as its name, identifier string, and version.

Mac OS X includes several slideshow screen savers you can use as templates for creating your own. These screen savers are located in `/System/Library/Screen Savers`. You should put your own slideshows in either `/Library/Screen Savers` or in the `~/Library/Screen Savers` directory of a user.

Programmatic Screen Savers

A programmatic screen saver is one that continuously generates content to appear on the screen. You can use this type of screen saver to create animations or to create a screen saver with user-configurable options. The bundle for a programmatic screen saver ends with the `.saver` extension.

You create programmatic screen savers using Cocoa and the Objective-C language. Specifically, you create a custom subclass of ScreenSaverView that provides the interface for displaying the screen saver content and options. The information property list of your bundle provides the system with the name of your custom subclass.

For information on creating programmatic screen savers, see *Screen Saver Framework Reference*.

Important: With the transition to Intel-based processors, developers should always create universal binaries for program-based screensavers written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Services

Services are not separate programs that you write; instead, they are features exported by your application for the benefit of other applications. Services let you share the resources and capabilities of your application with other applications in the system.

Services typically act on the currently selected data. Upon initiation of a service, the application that holds the selected data places it on the pasteboard. The application whose service was selected then takes the data, processes it, and puts the results (if any) back on the pasteboard for the original application to retrieve. For example, a user might select a folder in the Finder and choose a service that compresses the folder contents and replaces them with the compressed version. Services can represent one-way actions as well. For example, a service could take the currently selected text in a window and use it to create the content of a new email message.

For information on how to implement services in your Cocoa application, see *System Services*. For information on how to implement services in a Carbon application, see *Setting Up Your Carbon Application to Use the Services Menu*.

Preference Panes

Preference panes are used primarily to modify system preferences for the current user. Preference panes are implemented as plug-ins and installed in `/Library/PreferencePanes` on the user's system. Application developers can also take advantage of these plug-ins to manage per-user application preferences; however, most applications manage preferences using the code provided by the application environment.

You might need to create preference panes if you create:

- Hardware devices that are user-configurable
- Systemwide utilities, such as virus protection programs, that require user configuration

If you are an application developer, you might want to reuse preference panes intended for the System Preferences application or use the same model to implement your application preferences.

Because the interfaces are based on Objective-C, you write preference panes primarily using Cocoa. For more information, see *Preference Panes*.

Important: With the transition to Intel-based processors, developers should always create universal binaries for preference panes. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Web Content

Mac OS X supports a variety of techniques and technologies for creating web content. Dynamic websites and web services offer web developers a way to deliver their content quickly and easily.

In addition to "WebObjects" (page 24) and "Dashboard Widgets" (page 29), the following sections list ways to deliver web content in Mac OS X. For more information about developing web content, see *Getting Started with Internet and Web*.

Dynamic Websites

Mac OS X provides support for creating and testing dynamic content in web pages. If you are developing CGI-based web applications, you can create websites using a variety of scripting technologies, including Perl and PHP. A complete list of scripting technologies is provided in "Scripts" (page 34). You can also create and deploy more complex web applications using JBoss, Tomcat, and WebObjects. To deploy your webpages, use the built-in Apache web server.

Safari, Apple's web browser, provides standards-compliant support for viewing pages that incorporate numerous technologies, including HTML, XML, XHTML, DOM, CSS, Java, and JavaScript. You can also use Safari to test pages that contain multimedia content created for QuickTime, Flash, and Shockwave.

Sherlock Channels

The Sherlock application is a host for Sherlock channels. A Sherlock channel is a developer-created module that combines web services with an Aqua interface to provide a unique way for users to find information. You can use Sherlock channels to combine related, but different, types of information in one window. For example, the Sherlock movie channel displays information about nearby theaters and movie show times but also lets you view movie previews, posters, and a synopsis of the movie.

Sherlock channels are written using XML with JavaScript or XQuery code and are suited for:

- Web application designers who want to combine existing web services to provide added value for users
- Developers wanting to provide a user interface to their own web services

Because Sherlock channels are based on XML, JavaScript, and XQuery, you do not compile them as you would other types of programs. Xcode provides editing support for Sherlock channels, and Interface Builder provides support for creating a channel's window layout. However, to test your channel, you must install it on a web server.

For information on how to create a Sherlock channel, see *Sherlock Channels*.

SOAP and XML-RPC

The Simple Object Access Protocol (SOAP) is an object-oriented protocol that defines a way for programs to communicate over a network. XML-RPC is a protocol for performing remote procedure calls between programs. In Mac OS X, you can create clients that use these protocols to gather information from web services across the Internet. To create these clients, you use technologies such as PHP, JavaScript, AppleScript, and Cocoa.

If you want to provide your own web services in Mac OS X, you can use WebObjects or implement the service using the scripting language of your choice. You then post your script code to a web server, give clients a URL, and publish the message format your script supports.

For information on how to create client programs using AppleScript, see *XML-RPC and SOAP Programming Guide*. For information on how to create web services, see *WebObjects Web Services Programming Guide*.

Command-Line Tools

Command-line tools are simple programs that manipulate data using a text-based interface. These tools do not use windows, menus, or other user interface elements traditionally associated with applications. Instead, they run from the command-line environment of the Terminal application. Command-line tools require less explicit knowledge of the system to develop and because of that are often simpler to write than many other types of applications. However, command-line tools usually serve a more technically savvy crowd who are familiar with the conventions and syntax of the command-line interface.

Xcode supports the creation of command-line tools from several initial code bases. For example, you can create a simple and portable tool using standard C or C++ library calls, or a more Mac OS X–specific tool using frameworks such as Core Foundation, Core Services, or Cocoa Foundation.

Important: With the transition to Intel-based processors, developers should always create universal binaries for command-line tools written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Command-line tools are ideal for implementing simple programs quickly. You can use them to implement low-level system or administrative tools that do not need (or cannot have) a graphical user interface. For example, a system administrator might use command-line tools to gather status information from an Xserve system. You might also use them to test your program's underlying code modules in a controlled environment.

Note: Daemons are a special type of command-line program that run in the background and provide services to system and user-level programs. Developing daemons is not recommended, or necessary, for most developers.

Login Items

Login items are special programs that launch other programs or perform one-time operations during startup and login periods. Login items are typically used to launch daemons or run other programs that provide valuable services to system or user-level programs. A login item can also be a script that performs some periodic maintenance task, such as checking the hard drive for corrupted information.

Login items should not be confused with the startup items found in Accounts system preferences. Unlike startup items, login items are not user-configurable. Login items are usually provided directly by Mac OS X or are installed specially by a program that requires the target services to be available. Once installed, their services become available to all users and cannot be disabled without help from the program that installed them.

Few developers should ever need to create login items. Login items are reserved for the special case where you need to guarantee the availability of a particular service. For example, Mac OS X provides a login item to launch the DNS daemon. Similarly, a virus-detection program might install a login item to launch a daemon that monitors the system for virus-like activity. Both of these operations provide services that go beyond the scope of a single program to provide valuable services to the system and all the users on it.

For more information about login items, see *System Startup Programming Topics*.

Important: With the transition to Intel-based processors, developers should always create universal binaries for login items written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Scripts

A script is a set of text commands that are interpreted at runtime and turned into a sequence of actions. Most scripting languages provide high-level features that make it easy to implement complex workflows very quickly. Scripting languages are often very flexible, letting you call other programs and manipulate the data they return. Some scripting languages are also portable across platforms, so that you can use your scripts anywhere.

Table 2-1 lists many of the scripting languages supported by Mac OS X along with a description of the strengths of each language.

Table 2-1 Scripting language summary

Script language	Description
AppleScript	An English-based language for controlling scriptable applications in Mac OS X. Use it to tie together applications involved in a custom workflow or repetitive job. You can also use AppleScript Studio to create standalone applications whose code consists primarily of scripts. See AppleScript Documentation for more information.
`bash`	A Bourne-compatible shell script language used to build programs on UNIX-based systems.
`csh`	The C shell script language used to build programs on UNIX-based systems.
Perl	A general-purpose scripting language supported on many platforms. It comes with an extensive set of features suited for text parsing and pattern matching and also has some object-oriented features. See http://www.perl.org/ for more information.
PHP	A cross-platform, general-purpose scripting language that is especially suited for web development. See http://www.php.net/ for more information.
Python	A general-purpose, object-oriented scripting language implemented for many platforms. See http://www.python.org/ for more information.
Ruby	A general-purpose, object-oriented scripting language implemented for many platforms. See http://www.ruby-lang.org/ for more information.
`sh`	The Bourne shell script language used to build programs on UNIX-based systems.
Tcl	Tool Command Language. A general-purpose language implemented for many platforms. It is often used to create graphical interfaces for scripts. See http://www.tcl.tk/ for more information.
`tcsh`	A variant of the C shell script language used to build programs on UNIX-based systems.

Script language	Description
zsh	The Z shell script language used to build programs on UNIX-based systems.

For introductory material on using the command line, see "Command Line Primer" (page 89).

Scripting Additions for AppleScript

A scripting addition is a way to deliver additional functionality for AppleScript scripts. It extends the basic AppleScript command set by adding systemwide support for new commands or data types. Developers who need features not available in the current command set can use scripting additions to implement those features and make them available to all programs. For example, one of the built-in scripting additions extends the basic file-handling commands to support the reading and writing of file contents from an AppleScript script.

For information on how to create a scripting addition, see Technical Note TN1164, "Native Scripting Additions."

Important: With the transition to Intel-based processors, developers should always create universal binaries for scripting additions written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Kernel Extensions

Most developers have little need to create kernel extensions. Kernel extensions are code modules that load directly into the kernel process space and therefore bypass the protections offered by the Mac OS X core environment. The situations in which you might need a kernel extension are the following:

- Your code needs to handle a primary hardware interrupt.
- The client of your code is inside the kernel.
- A large number of applications require a resource your code provides. For example, you might implement a file-system stack using a kernel extension.
- Your code has special requirements or needs to access kernel interfaces that are not available in the user space.

Kernel extensions are typically used to implement new network stacks or file systems. You would not use kernel extensions to communicate with external devices such as digital cameras or printers. (For information on communicating with external devices, see "Device Drivers" (page 36).)

Note: Beginning with Mac OS X version 10.4, the design of the kernel data structures is changing to a more opaque access model. This change makes it possible for kernel developers to write nonfragile kernel extensions—that is, kernel extensions that do not break when the kernel data structures change. Developers are highly encouraged to use the new API for accessing kernel data structures.

For information about writing kernel extensions, see *Kernel Programming Guide*.

Important: With the transition to Intel-based processors, developers should always create universal binaries for kernel extensions. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Device Drivers

Device drivers are a special type of kernel extension that enable Mac OS X to communicate with all manner of hardware devices, including mice, keyboards, and FireWire drives. Device drivers communicate hardware status to the system and facilitate the transfer of device-specific data to and from the hardware. Mac OS X provides default drivers for many types of devices, but these may not meet the needs of all developers.

Although developers of mice and keyboards may be able to use the standard drivers, many other developers require custom drivers. Developers of hardware such as scanners, printers, AGP cards, and PCI cards typically have to create custom drivers for their devices. These devices require more sophisticated data handling than is usually needed for mice and keyboards. Hardware developers also tend to differentiate their hardware by adding custom features and behavior, which makes it difficult for Apple to provide generic drivers to handle all devices.

Apple provides code you can use as the basis for your custom drivers. The I/O Kit provides an object-oriented framework for developing device drivers using C++. For information on developing device drivers, see *I/O Kit Fundamentals*.

Important: With the transition to Intel-based processors, developers should always create universal binaries for device drivers. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

System-Level Technologies

This chapter summarizes the fundamental system technologies and facilities that are available to developers in Mac OS X. These services include graphics and rendering support, file-system support, network support, and other system facilities that provide the core of the system. It does not describe user-level technologies, such as Exposé, unless there is some aspect of the technology that requires developer involvement.

If you are new to developing Mac OS X software, you should read through this chapter at least once to understand the available technologies and how you might use them in your software. Even experienced developers should revisit this chapter periodically to remind themselves of the available technologies.

Core OS

The core environment of Mac OS X provides a stable foundation on which to develop software. All other technologies in Mac OS X are built on top of the technologies that reside in this layer of the system. Although you don't need to understand everything that goes on at this level, you should be familiar with some of the basic concepts that derive from this level.

Darwin

Beneath the appealing, easy-to-use interface of Mac OS X is a rock-solid, UNIX-based foundation called Darwin that is engineered for stability, reliability, and performance. Darwin integrates a number of technologies, most importantly Mach 3.0, operating-system services based on FreeBSD 5, high-performance networking facilities, and support for multiple, integrated file systems. Because the design of Darwin is highly modular, you can dynamically add such things as device drivers, networking extensions, and new file systems.

For more information about Darwin, see Darwin Documentation.

Mach

Mach is at the heart of Darwin because it performs some of the most critical functions of an operating system. Much of what Mach provides is transparent to applications. It manages processor resources such as CPU usage and memory, handles scheduling, enforces memory protection, and implements a messaging-centered infrastructure for untyped interprocess communication, both local and remote. Mach provides many important advantages to Macintosh computing:

- **Protected memory**. The stability of an operating system should not depend on all executing applications being good citizens. Even a well-behaved process can accidentally write data into the address space of the system or another process, which can result in the loss or corruption of data or even precipitate system crashes. Mach ensures that an application cannot write in another application's memory or in the operating system's memory. By walling off applications from each other and from system processes, Mach makes it virtually impossible for a single poorly behaved application to damage the rest of the system. Best of all, if an application crashes as the result of its own misbehavior, the crash affects only that application and not the rest of the system.
- **Preemptive multitasking**. With Mach, processes share the CPU efficiently. Mach watches over the computer's processor, prioritizing tasks, making sure activity levels are at the maximum, and ensuring that every task gets the resources it needs. It uses certain criteria to decide how important a task is and therefore how much time to allocate to it before giving another task its turn. Your process is not dependent on another process yielding its processing time.
- **Advanced virtual memory**. In Mac OS X, virtual memory is "on" all the time. The Mach virtual memory system gives each process its own private virtual address space. For 32-bit applications, this virtual address space is 4 GB. For 64-bit applications, the theoretical maximum is approximately 18 exabytes, or 18 billion billion bytes. Mach maintains address maps that control the translation of a task's virtual addresses into physical memory. Typically only a portion of the data or code contained in a task's virtual address space resides in physical memory at any given time. As pages are needed, they are loaded into physical memory from storage. Mach augments these semantics with the abstraction of memory objects. Named memory objects enable one task (at a sufficiently low level) to map a range of memory, unmap it, and send it to another task. This capability is essential for implementing separate execution environments on the same system.
- **Real-time support**. This feature guarantees low-latency access to processor resources for time-sensitive media applications.

Mach also enables cooperative multitasking, preemptive threading, and cooperative threading.

BSD

Integrated with Mach is a customized version of the BSD operating system (currently FreeBSD 5). Darwin's implementation of BSD includes much of the POSIX API, which is available from the application layers of the system. BSD serves as the basis for the file systems and networking facilities of Mac OS X. In addition, it provides several programming interfaces and services, including:

- The process model (process IDs, signals, and so on)
- Basic security policies such as file permissions and user and group IDs
- Threading support (POSIX threads)
- Networking support (BSD sockets)

X11

For UNIX developers, the X11 windowing system is now built in to Mac OS X starting with version 10.3. This windowing system is used by many UNIX applications to draw windows, controls, and other elements of a graphical user interface. The Mac OS X implementation of X11 uses the Quartz drawing environment to give its windows a native Mac OS X feel. This integration also makes it possible to display X11 windows alongside windows from native applications written in Carbon and Cocoa.

Device-Driver Support

For development of device drivers, Darwin offers an object-oriented framework called the I/O Kit. The I/O Kit facilitates the creation of drivers for Mac OS X and provides much of the infrastructure that they need. It is written in a restricted subset of C++. Designed to support a range of device families, the I/O Kit is both modular and extensible.

Device drivers created with the I/O Kit acquire several important features:

- True plug and play
- Dynamic device management (“hot plugging”)
- Power management (both desktops and portables)

If your device conforms to standard specifications, such as those for mice, keyboards, audio input devices, modern MIDI devices, and so on, it should just work when you plug it in. If your device doesn’t conform to a published standard, you can use the I/O Kit resources to create a custom driver to meet your needs. Devices such as AGP cards, PCI cards, scanners, and printers usually require custom drivers or other support software in order to work with Mac OS X.

For information on creating device drivers, see *I/O Kit Device Driver Design Guidelines*.

File-System Support

The file-system component of Darwin is based on extensions to BSD and an enhanced Virtual File System (VFS) design. The file-system component includes the following features:

- Permissions on removable media. This feature is based on a globally unique ID registered for each connected removable device (including USB and FireWire devices) in the system.
- Access control lists (available in Mac OS X version 10.4 and later)
- URL-based volume mount, which enables users (via a Finder command) to mount such things as AppleShare and web servers
- Unified buffer cache, which consolidates the buffer cache with the virtual-memory cache
- Long filenames (255 characters or 755 bytes, based on UTF-8)
- Support for hiding filename extensions on a per-file basis
- Journaling of all file-system types to aid in data recovery after a crash

Because of its multiple application environments and the various kinds of devices it supports, Mac OS X handles file data in many standard volume formats. Table 3-1 lists the supported formats.

Table 3-1 Supported local volume formats

Volume format	Description
Mac OS Extended Format	Also called HFS (hierarchical file system) Plus, or HFS+. This is the default root and booting volume format in Mac OS X. This extended version of HFS optimizes the storage capacity of large hard disks by decreasing the minimum size of a single file.
Mac OS Standard Format	Also called hierarchical file system, or HFS. This is the volume format in Mac OS systems prior to Mac OS 8.1. HFS (like HFS+) stores resources and data in separate forks of a file and makes use of various file attributes, including type and creator codes.
UFS	UNIX File System is a flat (that is, single-fork) disk volume format, based on the BSD FFS (Fast File System), that is similar to the standard volume format of most UNIX operating systems; it supports POSIX file-system semantics, which are important for many server applications.
UDF	Universal Disk Format, used for optical disks, including most types of writable CDs and DVDs. Mac OS X v10.4 supports the UDF 1.5 specification, which you can find at http://www.osta.org.
ISO 9660	The standard format for CD-ROM volumes.
NTFS	The NT File System, used by Windows computers. Mac OS X can read NTFS-formatted volumes but cannot write to them.
MS-DOS (FAT)	Mac OS X supports the FAT file systems used by many Windows computers. It can read and write FAT-formatted volumes.

HFS+ volumes support aliases, symbolic links, and hard links, whereas UFS volumes support symbolic links and hard links but not aliases. Although an alias and a symbolic link are both lightweight references to a file or directory elsewhere in the file system, they are semantically different in significant ways. For more information, see "Aliases and Symbolic Links" in *File System Overview*.

Note: Mac OS X does not support stacking in its file-system design.

Because Mac OS X is intended to be deployed in heterogeneous networks, it also supports several network file-sharing protocols. Table 3-2 lists these protocols.

Table 3-2 Supported network file-sharing protocols

File protocol	Description
AFP client	Apple Filing Protocol, the principal file-sharing protocol in Mac OS 9 systems (available only over TCP/IP transport).
NFS client	Network File System, the dominant file-sharing protocol in the UNIX world.
WebDAV	Web-based Distributed Authoring and Versioning, an HTTP extension that allows collaborative file management on the web.

File protocol	Description
SMB/CIFS	SMB/CIFS, a file-sharing protocol used on Windows and UNIX systems.

Network Support

Mac OS X is one of the premier platforms for computing in an interconnected world. It supports the dominant media types, protocols, and services in the industry as well as differentiated and innovative services from Apple.

The Mac OS X network protocol stack is based on BSD. The extensible architecture provided by network kernel extensions, summarized in "Networking Extensions" (page 44), facilitates the creation of modules implementing new or existing protocols that can be added to this stack.

Standard Network Protocols

Mac OS X provides built-in support for a large number of network protocols that are standard in the computing industry. Table 3-3 summarizes these protocols.

Table 3-3 Network protocols

Protocol	Description
802.1x	802.1x is a protocol for implementing port-based network access over wired or wireless LANs. It supports a wide range of authentication methods, including TLS, TTLS, LEAP, MDS, and PEAP (MSCHAPv2, MD5, GTC).
DHCP and BOOTP	The Dynamic Host Configuration Protocol and the Bootstrap Protocol automate the assignment of IP addresses in a particular network.
DNS	Domain Name Services is the standard Internet service for mapping host names to IP addresses.
FTP and SFTP	The File Transfer Protocol and Secure File Transfer Protocol are two standard means of moving files between computers on TCP/IP networks. (SFTP support was added in Mac OS X version 10.3.)
HTTP and HTTPS	The Hypertext Transport Protocol is the standard protocol for transferring webpages between a web server and browser. Mac OS X provides support for both the insecure and secure versions of the protocol.
LDAP	The Lightweight Directory Access Protocol lets users locate groups, individuals, and resources such as files and devices in a network, whether on the Internet or on a corporate intranet.
NBP	The Name Binding Protocol is used to bind processes across a network.
NTP	The Network Time Protocol is used for synchronizing client clocks.
PAP	The Printer Access Protocol is used for spooling print jobs and printing to network printers.

Protocol	Description
PPP	For dialup (modem) access, Mac OS X includes PPP (Point-to-Point Protocol). PPP support includes TCP/IP as well as the PAP and CHAP authentication protocols.
PPPoE	The Point-to-Point Protocol over Ethernet protocol provides an Ethernet-based dialup connection for broadband users.
S/MIME	The Secure MIME protocol supports encryption of email and the attachment of digital signatures to validate email addresses. (S/MIME support was added in Mac OS X version 10.3.)
SLP	Service Location Protocol is designed for the automatic discovery of resources (servers, fax machines, and so on) on an IP network.
SOAP	The Simple Object Access Protocol is a lightweight protocol for exchanging encapsulated messages over the web or other networks.
SSH	The Secure Shell protocol is a safe way to perform a remote login to another computer. Session information is encrypted to prevent unauthorized snooping of data.
TCP/IP and UDP/IP	Mac OS X provides two transmission-layer protocols, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), to work with the network-layer Internet Protocol (IP). (Mac OS X 10.2 and later includes support for IPv6 and IPSec.)
XML-RPC	XML-RPC is a protocol for sending remote procedure calls using XML over the web.

Apple also implements a number of file-sharing protocols; see Table 3-2 (page 40) for a summary of these protocols.

Legacy Network Services and Protocols

Apple includes the following legacy network products in Mac OS X to ease the transition from earlier versions of the Mac OS.

- AppleTalk is a suite of network protocols that is standard on the Macintosh and can be integrated with other network systems. Mac OS X includes minimal support for compatibility with legacy AppleTalk environments and solutions.
- Open Transport implements industry-standard communications and network protocols as part of the I/O system. It helps developers incorporate networking services in their applications without having to worry about communication details specific to any one network.

Existing applications can continue to use these technologies. However, if you are developing new applications, you should use newer networking technologies such as CFNetwork.

Network Technologies

Mac OS X supports the network technologies listed in Table 3-4.

Table 3-4 Network technology support

Technology	Description
Ethernet 10/100Base-T	For the Ethernet ports built into every new Macintosh.
Ethernet 1000Base-T	Also known as Gigabit Ethernet. For data transmission over fiber-optic cable and standardized copper wiring.
Jumbo Frame	This Ethernet format uses 9 KB frames for interserver links rather than the standard 1.5 KB frame. Jumbo Frame decreases network overhead and increases the flow of server-to-server and server-to-application data. Jumbo frames are supported in Mac OS X version 10.3 and later. Systems running Mac OS X versions 10.2.4 to 10.3 can use jumbo frames only on third-party Ethernet cards that support them.
Serial	Supports modem and ISDN capabilities.
Wireless	Supports the 802.11b and 802.11g wireless network technology using AirPort and AirPort Extreme.

Routing and Multihoming

Mac OS X is a powerful and easy-to-use desktop operating system but can also serve as the basis for powerful server solutions. Some businesses or organizations have small networks that could benefit from the services of a router, and Mac OS X offers IP routing support for just these occasions. With IP routing, a Mac OS X computer can act as a router or even as a gateway to the Internet. The Routing Information Protocol (RIP) is used in the implementation of this feature.

Mac OS X also allows multihoming and IP aliasing. With multihoming, a computer host is physically connected to multiple data links that can be on the same or different networks. IP aliasing allows a network administrator to assign multiple IP addresses to a single network interface. Thus one computer running Mac OS X can serve multiple websites by acting as if it were multiple servers.

Zero-Configuration Networking

Introduced in Mac OS X version 10.2, Bonjour is Apple's implementation of zero-configuration networking. Bonjour enables the dynamic discovery of computer services over TCP/IP networks without the need for any complex user configuration of the associated hardware. Bonjour helps to connect computers and other electronic devices by providing a mechanism for them to advertise and browse for network-based services. See "Bonjour" (page 66) for more information.

NetBoot

NetBoot is most often used in school or lab environments where the system administrator needs to manage the configuration of multiple computers. NetBoot computers share a single System folder, which is installed on a centralized server that the system administrator controls. Users store their data in home directories on the server and have access to a common Applications folder, both of which are also commonly installed on the server.

To support NetBoot, applications must be able to run from a shared, locked volume and write a user's personal data to a different volume. Preferences and user-specific data should always be stored in the Preferences folder of the user's home directory. Users should also be asked where they want to save their data, with the user's Documents folder being the default location. Applications must also remember that multiple users may run the application simultaneously.

See Technical Note TN1151, "Creating NetBoot Server–Friendly Applications," for additional information. For information on how to write applications that support multiple simultaneous users, see *Multiple User Environments*.

Personal Web Sharing

Personal Web Sharing allows users to share information with other users on an intranet, no matter what type of computer or browser they are using. Basically, it lets users set up their own intranet site. Apache, the most popular web server on the Internet, is integrated as the system's HTTP service. The host computer on which the Personal Web Sharing server is running must be connected to a TCP/IP network.

Networking Extensions

Darwin offers kernel developers a technology for adding networking capabilities to the operating system: network kernel extensions (NKEs). The NKE facility allows you to create networking modules and even entire protocol stacks that can be dynamically loaded into the kernel and unloaded from it. NKEs also make it possible to configure protocol stacks automatically.

NKE modules have built-in capabilities for monitoring and modifying network traffic. At the data-link and network layers, they can also receive notifications of asynchronous events from device drivers, such as when there is a change in the status of a network interface.

For information on how to write an NKE, see *Network Kernel Extensions (legacy)*.

Network Diagnostics

Introduced in Mac OS X version 10.4, network diagnostics is a way of helping the user solve network problems. Although modern networks are generally reliable, there are still times when network services may fail. Sometimes the cause of the failure is beyond the ability of the desktop user to fix, but sometimes the problem is in the way the user's computer is configured. The network diagnostics feature provides a diagnostic application to help the user locate problems and correct them.

If your application encounters a network error, you can use the new diagnostic interfaces of CFNetwork to launch the diagnostic application and attempt to solve the problem interactively. You can also choose to report diagnostic problems to the user without attempting to solve them.

For more information on using this feature, see the header files of CFNetwork.

Velocity Engine

Velocity Engine is a hardware-based vector computation unit built into the G4 and G5 processors. Support for Velocity Engine is another important feature of Mac OS X. Velocity Engine boosts the performance of any application exploiting data parallelism, such as those that perform 3D graphic imaging, image processing, video processing, audio compression, and software-based cell telephony. Quartz and QuickTime incorporate Velocity Engine capabilities, thus any application using these APIs can tap into Velocity Engine without making any changes. The Xcode Tools include a C/C++ compiler with Velocity Engine support so that you can create new applications that take full advantage of Velocity Engine.

Mac OS X version 10.3 includes a new framework for accelerating complex operations by taking advantage of Velocity Engine. The Accelerate framework is an umbrella framework that wraps the existing vecLib framework and the new vImage framework. The vecLib framework contains vector-optimized routines for doing digital signal processing, linear algebra, and other computationally expensive mathematical operations. (This framework is also a top-level framework for applications running on earlier versions of Mac OS X.) The vImage framework supports the visual realm, adding routines for morphing, alpha-channel processing, and other image-buffer manipulations.

See Performance Documentation for information about the Velocity Engine–accelerated frameworks.

Java Support

The following sections outline the support provided by Mac OS X for creating Java-based programs.

Note: The developer documentation on the Apple website contains an entire section devoted to Java. There you can find detailed information on the Java environment and accompanying technologies for operating in Mac OS X. For an introduction to the Java environment, see *Getting Started with Java*. The Fundamentals section of Java Documentation also contains information about the latest Java releases for Mac OS X.

The Java Environment

The libraries, JAR files, and executables for the Java application environment are located in the `/System/Library/Frameworks/JavaVM.framework` directory. The Java application environment has three major components:

- A development environment, comprising the Java compiler (`javac`) and debugger (`jdb`) as well as other tools, including `javap`, `javadoc`, and `appletviewer`. You can also build Java applications using Xcode.
- A runtime environment consisting of Sun's high-performance HotSpot Java virtual machine, the "just-in-time" (JIT) bytecode compiler, and several basic packages, including `java.lang`, `java.util`, `java.io`, and `java.net`.
- An application framework containing the classes necessary for building a Java application. This framework contains the Abstract Windowing Toolkit (`java.awt`) and Swing (`javax.swing`) packages, among others. These packages provide user interface components, basic drawing capabilities, a layout manager, and an event-handling mechanism.

Like Carbon and Cocoa applications, a Java application can be distributed as a double-clickable bundle. The Jar Bundler tool takes your Java packages and produces a Mac OS X bundle. This tool is installed along with Xcode and the rest of the Apple developer tools on the Xcode Tools CD.

If you want to run your Java application from the command line, you can use the `java` command. To launch a Java application from another program, use the system `exec` call or the Java `Runtime.exec` method. To run applets, embed the applet into an HTML page and open the page in Safari.

Java and Other Application Environments

Mac OS X lets you write Java applications that take advantage of native technologies such as Carbon, Cocoa, and QuickTime. For example, Sun's Java Native Interface (JNI) lets your Java programs interact with other system frameworks, such as the Carbon framework.

The Cocoa application environment includes Java packages corresponding to the Foundation and Application Kit frameworks. These packages allow you to develop a Cocoa application using Java as the development language. For an example of how to create Java applications with Cocoa, see *Cocoa Tutorial for Java Programmers*.

Versions of QuickTime for Java are available for both the Mac OS X and Windows platforms, allowing you to create cross-platform Java applications using QuickTime. For information on creating QuickTime applications with Java, see QuickTime Documentation.

64-Bit Support

In Mac OS X version 10.4 and later, the Xcode Tools support the compilation, linking, and debugging of 64-bit binaries using C or C++. In addition to the tools support, the system also includes 64-bit versions of `libSystem.dylib`, which contains much of the C standard library code, and the Accelerate framework.

Support for 64-bit computing makes it possible to operate on large data sets more efficiently. For more information about creating 64-bit applications, see *64-Bit Transition Guide*.

Graphics, Imaging, and Multimedia

The graphics and multimedia capabilities of Mac OS X set it apart from other operating systems. The technologies that drive these capabilities can easily be integrated into your applications and are described in the following sections.

Quartz

Quartz is at the heart of the Mac OS X graphics and windowing environment. It provides rendering support for 2D shapes and text and combines a rich imaging model with on-the-fly rendering, compositing, and anti-aliasing of content. Quartz also implements the windowing system for Mac OS X and provides low-level services such as event routing and cursor management.

Quartz comprises both a client API (Quartz 2D) and a window server (Quartz Compositor). Applications use the client API to draw primitive shapes and text in their windows. Quartz Compositor is a lightweight window server that provides essential services to clients (through the Quartz client API) but does not perform any rendering itself.

The Quartz 2D client API is implemented as part of the Application Services umbrella framework (`ApplicationServices.framework`), which is what you include in your projects when you want to use Quartz. This umbrella framework includes the Core Graphics framework (`CoreGraphics.framework`), which defines the Quartz 2D interfaces, types, and constants you use in your applications.

Quartz also includes an integrated component called Quartz Services, which provides direct access to some low-level features of the window server. Access to configuration information for the currently connected display hardware lets you capture a display for exclusive use and adjust display attributes, such as resolution, pixel depth, and refresh rate. Quartz Services also provides some support for operating a Mac OS X system remotely.

For information about Quartz, see *Quartz 2D Programming Guide*.

Digital Paper Metaphor

The Quartz imaging architecture is based on a digital paper metaphor. In this case, the digital paper is PDF, which is also the internal model used by Quartz to store rendered content. Content stored in this medium has a very high fidelity and can be reproduced on many different types of devices, including displays, printers, and fax machines. This content can also be written to a PDF file and viewed by any number of applications that display the PDF format.

The PDF model gives application developers much more control over the final appearance of their content. PDF takes into account the application's choice of color space, fonts, image compression, and resolution. Vector artwork can be scaled and manipulated during rendering to implement unique effects, such as those that occur when the system transitions between users with the fast user switching feature.

Mac OS X also takes advantage of the flexibility of PDF in implementing some system features. For example, in addition to printing, the standard printing dialogs offer options to save a document as PDF, preview the document before printing, or transmit the document using a fax machine. The PDF used for all of these operations comes from the same source: the pages formatted for printing by the application's rendering code. The only difference is the device to which that content is sent.

Quartz 2D Features

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

- High-quality rendering on the screen
- Resolution independent UI support
- Anti-aliasing 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

Table 3-5 describes some of technical specifications for Quartz.

Table 3-5 Quartz technical specifications

Bit depth	A minimum bit depth of 16 bits for typical users. An 8-bit depth in full-screen mode is available for Classic applications, games, and other multimedia applications.
Minimum resolution	Supports 800 pixels by 600 pixels as the minimum screen resolution for typical users. A resolution of 640 x 480 is available for the iBook as well as for Classic applications, games, and other multimedia applications.
Velocity Engine	Quartz and QuickDraw both take advantage of Velocity Engine to boost performance.
Quartz Extreme	Quartz Extreme uses OpenGL for the entire Mac OS X desktop. Graphics calls now render in supported video hardware, freeing up the CPU for other tasks.

Quartz Compositor

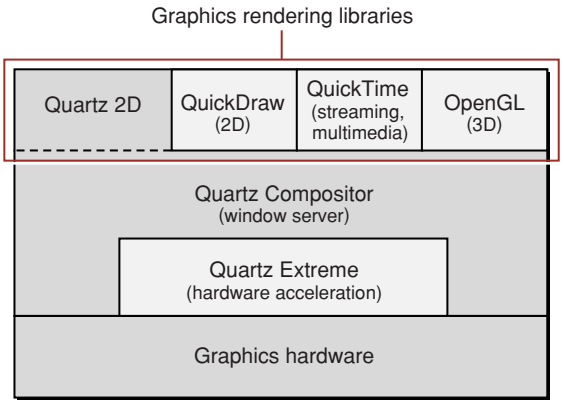
Quartz Compositor, the window server for Mac OS X, coordinates all of the low-level windowing behavior and enforces a fundamental uniformity in what appears on the screen. It manages the displays available on the user's system, interacting with the necessary device drivers. It also provides window management, event-routing, and cursor management behaviors.

In addition to window management, Quartz Compositor handles the compositing of all visible content on the user's desktop. Rather than composite a flat bitmap of each application window, Quartz Compositor supports transparency effects through the use of alpha channel information. The transparency information makes it possible to display drop shadows, cutouts, and other effects that add a more realistic and dimensional texture to the windows.

The performance of Quartz Compositor remains consistently high because of several factors. To improve window redrawing performance, Quartz Compositor supports buffered windows and the layered compositing of windows and window content. Thus, windows that are hidden behind opaque content are never composited. Quartz Compositor also incorporates Quartz Extreme, which speeds up rendering calls by handing them off to graphics hardware whenever possible.

Figure 3-1 shows the high-level relationships between Quartz Compositor and the rendering technologies available on Mac OS X. QuickTime and OpenGL have fewer dependencies on Quartz Compositor because they implement their own versions of certain windowing capabilities.

Figure 3-1 Quartz Compositor and the rendering APIs in Mac OS X



QuickTime

QuickTime is a powerful multimedia technology for manipulating, enhancing, and storing video, sound, animation, graphics, text, music, and even 360-degree virtual reality. It allows you to stream digital video, where the data stream can be either live or stored. QuickTime is a cross-platform technology, supporting Mac OS X, Mac OS 9, Windows 98, Windows Me, Windows 2000, and Windows XP. Using QuickTime, developers can perform actions such as the following:

- Open and play movie files
- Open and play audio files
- Display still images
- Translate still images from one format to another
- Compress audio, video, and still images
- Synchronize multiple media to a common timeline
- Capture audio and video from an external device
- Stream audio and video over a LAN or the Internet
- Create and display virtual reality objects and panoramas

For a long time, QuickTime has included programming interfaces for the C , C++, and Java languages. Beginning with Mac OS X v10.4, the QuickTime Kit provides an Objective-C based set of classes for managing QuickTime content. For more information on QuickTime Kit, see "QuickTime Kit" (page 71)

Supported Media Formats

QuickTime supports more than a hundred media types, covering a range of audio, video, image, and streaming formats. Table 3-6 lists some of the more common file formats it supports. For a complete list of supported formats, see the QuickTime product specification page at http://www.apple.com/quicktime/pro/specs.html.

Table 3-6 Partial list of formats supported by QuickTime

Image formats	PICT, BMP, GIF, JPEG, TIFF, PNG
Audio formats	AAC, AIFF, MP3, WAVE, uLaw
Video formats	AVI, AVR, DV, M-JPEG, MPEG-1, MPEG-2, MPEG-4, AAC, OpenDML, 3GPP, 3GPP2, AMC, H.264
Web streaming formats	HTTP, RTP, RTSP

Extending QuickTime

The QuickTime architecture is very modular. QuickTime includes media handler components for different audio and video formats. Components also exist to support text display, Flash media, and codecs for different media types. However, most applications do not need to know about specific components. When an application tries to open and play a specific media file, QuickTime automatically loads and unloads the needed components. Of course, applications can specify components explicitly for many operations.

You can extend QuickTime by writing your own component. You might write your own QuickTime component to support a new media type or to implement a new codec. You might also write components to support a custom video capture card. By implementing your code as a QuickTime component that you enable, other applications take advantage of your code and use it to support your hardware or media file formats. See "QuickTime Components" (page 28) for more information.

OpenGL

OpenGL is an industry-wide standard for developing portable three-dimensional (3D) graphics applications. It is specifically designed for games, animation, CAD/CAM, medical imaging, and other applications that need a rich, robust framework for visualizing shapes in two and three dimensions. The OpenGL API is one of the most widely adopted graphics API standards, which makes code written for OpenGL portable and consistent across platforms. The OpenGL framework (`OpenGL.framework`) in Mac OS X includes a highly optimized implementation of the OpenGL libraries that provides high-quality graphics at a consistently high level of performance.

OpenGL offers a broad and powerful set of imaging functions, including texture mapping, hidden surface removal, alpha blending (transparency), anti-aliasing, pixel operations, viewing and modeling transformations, atmospheric effects (fog, smoke, and haze), and other special effects. Each OpenGL command directs a drawing action or causes a special effect, and developers can create lists of these commands for repetitive effects. Although OpenGL is largely independent of the windowing characteristics of each operating system, the standard defines special glue routines to enable OpenGL to work in an operating system's windowing environment. The Mac OS X implementation of OpenGL implements these glue routines to enable operation with the Quartz Compositor.

For information about incorporating OpenGL calls into your application, see the documentation in Graphics & Imaging OpenGL Documentation.

OpenAL

Beginning with Mac OS X version 10.4, the system comes with the Open Audio Library (OpenAL) audio system installed. This interface is a cross-platform standard for delivering 3D audio in applications. OpenAL lets you implement high-performance positional audio in games and other programs that require high-quality audio output. Because it is a cross-platform standard, the applications you write using OpenAL on Mac OS X can be ported to run on many other platforms.

Apple's implementation of OpenAL is based on Core Audio, so it delivers high-quality sound and performance on Mac OS X systems. To use OpenAL in a Mac OS X application, include the OpenAL framework (`OpenAL.framework`) in your Xcode project. This framework includes header files whose contents conform to the OpenAL specification, which is described at http://www.openal.org.

Core Audio

The Core Audio frameworks of Mac OS X offer a sophisticated set of services for manipulating multichannel audio. Core Audio includes support for the following features (among others):

- Manipulating audio files
- Controlling audio devices
- Sequencing audio
- Adding effects
- Mixing audio channels
- Combining audio devices into one aggregate device
- Working with MIDI hardware and software

For information about the Core Audio components, see the information in Music & Audio Reference Library.

Core Image

Introduced in Mac OS X version 10.4, Core Image extends the basic graphics capabilities of the system to provide a framework for implementing complex visual behaviors in your application. Core Image uses GPU-based acceleration and 32-bit floating-point support to provide fast image processing and pixel-level accurate content. The plug-in based architecture lets you expand the capabilities of Core Image through the creation of image units, which implement the desired visual effects.

Core Image includes built-in image units that allow you to:

- Crop images
- Correct color, including perform white-point adjustments
- Apply color effects, such as sepia tone
- Blur or sharpen images
- Composite images

- Warp the geometry of an image by applying an affine transform or a displacement effect
- Generate color, checkerboard patterns, Gaussian gradients, and other pattern images
- Add transition effects to images or video
- Provide real-time control, such as color adjustment and support for sports, vivid, and other video modes
- Apply linear lighting effects, such as spotlight effects

You define custom image units using the classes of the Core Image framework. You can use both the built-in and custom image units in your application to implement special effects and perform other types of image manipulations. Image units take full advantage of Velocity Engine, Quartz, OpenGL, and QuickTime to optimize the processing of video and image data. Rasterization of the data is ultimately handled by OpenGL, which takes advantage of graphics hardware acceleration whenever it is available.

Core Image is part of the Quartz Core framework (`QuartzCore.framework`). For information about how to use Core Image or how to write custom image units, see *Core Image Programming Guide* and *Core Image Reference Collection*.

Core Video

Introduced in Mac OS X version 10.4, Core Video provides a modern foundation for delivering video in your applications. It creates a bridge between QuickTime and the GPU to deliver hardware-accelerated video processing. By offloading complex processing to the GPU, you can significantly increase performance and reduce the CPU load of your applications. Core Video also allows developers to apply all the benefits of Core Image to video, including filters and effects, per-pixel accuracy, and hardware scalability.

Core Video is part of the Quartz Core framework (`QuartzCore.framework`).

ColorSync

ColorSync is the color management system for Mac OS X. It provides essential services for fast, consistent, and accurate color calibration, proofing, and reproduction as well as an interface for accessing and managing systemwide color management settings. It also supports color calibration with hardware devices such as printers, scanners, and displays.

Beginning with Mac OS X version 10.3, the system provides improved support for ColorSync. In most cases, you do not need to call ColorSync functions at all. Quartz automatically uses ColorSync to manage pixel data when drawing on the screen. It also respects ICC (International Color Consortium) profiles and applies the system's monitor profile as the source color space. However, you might need to use ColorSync directly if you define a custom color management module (CMM), which is a component that implements color-matching, color-conversion, and gamut-checking services.

For information about the ColorSync API, see *ColorSync Manager Reference*.

DVD Playback

Mac OS X version 10.3 and later includes the DVD Playback framework for embedding DVD viewer capabilities into an application. In addition to playing DVDs, you can use the framework to control various aspects of playback, including menu navigation, viewer location, angle selection, and audio track selection. You can play back DVD data from disc or from a local `VIDEO_TS` directory.

For more information about using the DVD Playback framework, see the framework's header file.

Printing

Printing is a collection of APIs and system services available to all application environments. Drawing on the capabilities of Quartz, the printing system delivers a consistent human interface and makes shorter development cycles possible for printer vendors. It also provides applications with a high degree of control over the user interface elements in printing dialogs. Table 3-7 describes some other features of the Mac OS X printing system.

Table 3-7 Features of the Mac OS X printing system

Feature	Description
CUPS	The Common Unix Printing System (CUPS) provides the underlying support for printing. It is an open-source architecture used commonly by the UNIX community to handle print spooling and other low-level features.
Desktop printers	Beginning in Mac OS X version 10.3, desktop printers offer users a way to manage printing from the Dock or desktop. Users can print natively supported files (like PostScript and PDF) by dragging them to a desktop printer. Users can also manage print jobs.
Fax support	Beginning with Mac OS X version 10.3, users can fax documents directly from the Print dialog.
GIMP-Print drivers	Mac OS X version 10.3 and later includes drivers for many older printers through the print facility of the GNU Image Manipulation Program (GIMP).
Native PDF	Supports PDF as a native data type. Any application (except for Classic applications) can easily save textual and graphical data to device-independent PDF where appropriate. The printing system provides this capability from a standard printing dialog.
PostScript support	Mac OS X prints to PostScript Level 2–compatible and Level 3–compatible printers. Beginning with Mac OS X version 10.3, support is also provided to convert PostScript files directly to PDF.
Print preview	Provides a print preview capability in all environments, except in Classic. The printing system implements this feature by launching a PDF viewer application. This preview is color-managed by ColorSync.
Printer discovery	Printers implementing Bluetooth or Bonjour can be detected, configured, and added to printer lists automatically.

Feature	Description
Raster printers	Supports printing to raster printers in all environments, except in the Classic environment.
Speedy spooling	Beginning with Mac OS X version 10.3, applications that use PDF can submit PDF files directly to the printing system instead of spooling individual pages. This simplifies printing for applications that already store data as PDF.

For an overview of the printing architecture and how to support it, see *Mac OS X Printing System Overview*.

Text and Fonts

Mac OS X provides extensive support for typography through advanced Carbon and Cocoa APIs. These APIs let you control the fonts, layout, typesetting, text input, and text storage in your programs.

Apple Type Services

Apple Type Services (ATS) is the engine for the systemwide management, layout, and rendering of fonts. With ATS, users can have a single set of fonts distributed over different parts of the file system or even over a network. ATS makes the same set of fonts available to all clients. The centralization of font rendering and layout contributes to overall system performance by consolidating expensive operations such as synthesizing font data and rendering glyphs. ATS provides support for a wide variety of font formats, including TrueType, PostScript Type 1, and PostScript OpenType.

For more information about ATS, see *Managing Fonts: ATS*.

Cocoa Text

Cocoa provides advanced text-handling capabilities in the Application Kit framework. Based on ATSUI, the Cocoa text system provides a multilayered approach to implementing a full-featured text system using Objective-C. This layered approach lets you customize portions of the system that are relevant to your needs while using the default behavior for the rest of the system.

For an overview of the Cocoa text system, see *Text System Overview*.

Multilingual Text Engine

The Multilingual Text Engine (MLTE) is an API that provides Carbon-compliant Unicode text editing. MLTE replaces TextEdit and provides an enhanced set of features, including document-wide tabs, text justification, built-in scroll bar handing, built-in printing support, inline input, multiple levels of undo, support for more than 32 KB of text, and support for Apple Type Services. This API is designed for developers who want to incorporate a full set of text editing features into their applications but do not want to worry about managing the text layout or typesetting.

For more information about MLTE, see *Handling Unicode Text Editing With MLTE*.

Apple Type Services for Unicode Imaging

Apple Type Services for Unicode Imaging (ATSUI) is the technology behind all text drawing in Mac OS X. ATSUI gives developers precise control over text layout features and supports high-end typography. It is intended for developers of desktop publishing applications or any application that requires the precise manipulation of text.

For more information about ATSUI, see *Rendering Unicode Text With ATSUI*.

QuickDraw

QuickDraw is a legacy technology adapted from earlier versions of the Mac OS that lets you construct, manipulate, and display two-dimensional shapes, pictures, and text. Because it is a legacy technology, the use of QuickDraw in any new projects is discouraged.

If your code currently uses QuickDraw, you should begin converting it to Quartz 2D as soon as possible. The QuickDraw API includes interfaces for getting a Quartz graphics context from a `GrafPort` structure. You can migrate away from QuickDraw gradually without radically affecting your product development cycles, by integrating Quartz rendering calls inside your QuickDraw code.

Interprocess Communication

Mac OS X supports numerous technologies for interprocess communication (IPC). The following sections describe the available technologies.

Distributed Objects for Cocoa

Cocoa distributed objects provide a transparent mechanism that allows different applications (or threads in the same application) to communicate on the same computer or across the network. The implementation of distributed objects lets you focus on the data being transferred rather than the connection. As a result, implementing distributed objects takes less time than most other IPC mechanisms; however, this ease of implementation comes at the cost of performance. Distributed objects are typically not as efficient as many other techniques.

For information on how to use distributed objects in your Cocoa application, see *Distributed Objects*.

Apple Events

An Apple event is a high-level semantic event that an application can send to itself, to other applications on the same computer, or to applications on a remote computer. Apple events are the primary technology used for scripting and interapplication communication in Mac OS X. Applications can use Apple events to request services and information from other applications. To supply services, you define objects in your application that can be accessed using Apple events and then provide Apple event handlers to respond to requests for those objects.

Apple events have a well-defined data structure that supports extensible, hierarchical data types. To make it easier for scripters and other developers to access it, your application should generally support the standard set of events defined by Apple. If you want to support additional features not covered by the standard suite, you can also define custom events as needed.

Apple events are part of the Application Services umbrella framework. For information on how to use Apple events, see *Apple Events Programming Guide*. See also *Apple Event Manager Reference* for information about the functions and constants used to create, send, and receive Apple events.

Distributed Notifications

A distributed notification is a message posted by any process to a per-computer notification center, which in turn broadcasts the message to any processes interested in receiving it. Included with the notification is the ID of the sender and an optional dictionary containing additional information. The distributed notification mechanism is implemented by the Core Foundation CFNotificationCenter object and by the Cocoa NSDistributedNotificationCenter class.

Distributed notifications are ideal for simple notification-type events. For example, a notification might communicate the status of a certain piece of hardware, such as the network interface or a typesetting machine. However, notifications should not be used to communicate critical information to a specific process. Although Mac OS X makes every effort possible, it does not guarantee the delivery of a notification to every registered receiver.

Distributed notifications are true notifications because there is no opportunity for the receiver to reply to them. There is also no way to restrict the set of processes that receive a distributed notification. Any process that registers for a given notification may receive it. Because distributed notifications use a string for the unique registration key, there is also a potential for namespace conflicts.

For information on Core Foundation support for distributed notifications, see *CFNotificationCenter Reference*. For information about Cocoa support for distributed notifications, see *Notification Programming Topics for Cocoa*.

BSD Notifications

Starting with Mac OS X version 10.3, applications can take advantage of a system-level notification API. This notification mechanism is defined in the `/usr/include/notify.h` system header. BSD notifications offer some advantages over the Core Foundation notification mechanism, including the following:

- Clients can receive BSD notifications through several different mechanisms, including Mach ports, signals, and file descriptors.
- BSD notifications are more lightweight and efficient than other notification techniques.
- BSD notifications can be coalesced if multiple notifications are received in quick succession.

You can add support for BSD notifications to any type of program, including Carbon and Cocoa applications. To view the man pages for the routines defined in `notify.h`, use the command `man 3 notify` from Terminal. The man pages are also included in the ADC Reference Library. For more information, see *Mac OS X Man Pages*.

Sockets, Ports, and Streams

Sockets and ports provide a portable mechanism for communicating between applications in Mac OS X. A socket represents one end of a communications channel between two processes either locally or across the network. A port is a channel between processes or threads on the local computer. Applications can set up sockets and ports to implement fast, efficient messaging between processes.

The Core Foundation framework includes abstractions for sockets (CFSocket/CFRunLoop) and ports (CFMessagePort). You can use CFSocket with CFRunLoop to multiplex data received from a socket with data received from other sources. This allows you to keep the number of threads in your application to an absolute minimum, which conserves system resources and thus aids performance. Core Foundation sockets are also much simpler to use than the raw socket interfaces provided by BSD. CFMessagePort provides similar features for ports.

If you are communicating using an established transport mechanism such as Bonjour or HTTP, a better way to transfer data between processes is with the Core Foundation or Cocoa stream interfaces. These interfaces work with CFNetwork to provide a stream-based way to read and write network data. Like sockets, streams and CFNetwork were designed with run loops in mind and operate efficiently in that environment.

CFSocket and its related functions are documented in *CFSocket Reference*. For information about Core Foundation streams, see *CFReadStream Reference* and *CFWriteStream Reference*. For information about Cocoa streams, see the description of the NSStream class in *Foundation Framework Reference*.

BSD Pipes

A pipe is a communications channel typically created between a parent and a child process when the child process is forked. Data written to a pipe is buffered and read in first-in, first-out (FIFO) order. You create unnamed pipes between a parent and child using the `pipe` function declared in `/usr/include/unistd.h`. This is the simplest way to create a pipe between two processes; the processes must, however, be related.

You can also create named pipes to communicate between any two processes. A named pipe is represented by a file in the file system called a FIFO special file. A named pipe must be created with a unique name known to both the sending and the receiving process.

Note: Make sure you give your named pipes appropriate names to avoid unwanted collisions caused by the presence of multiple simultaneous users.

Pipes are a convenient and efficient way to create a communications channel between related processes. However, in general use, pipes are still not as efficient as using CFStream. The run loop support offered by CFStream makes it a better choice when you have multiple connections or plan to maintain an open channel for an extended period of time.

The interfaces for CFStream are documented in *CFNetwork Programming Guide*.

Shared Memory

Shared memory is a region of memory that has been allocated by a process specifically for the purpose of being readable and possibly writable among several processes. Access to shared memory is controlled through POSIX semaphores, which implement a kind of locking mechanism. Shared memory has some distinct advantages over other forms of interprocess communication:

- Any process with appropriate permissions can read or write a shared memory region.
- Data is never copied. Each process reads the shared memory directly.
- Shared memory offers excellent performance.

The disadvantage of shared memory is that it is very fragile. When a data structure in a shared memory region becomes corrupt, all processes that refer to the data structure are affected. In most cases, shared memory regions should also be isolated to a single user session to prevent security issues. For these reasons, shared memory is best used only as a repository for raw data (such as pixels or audio), with the controlling data structures accessed through more conventional interprocess communication.

Mach Messaging

Mach port objects implement a standard, safe, and efficient construct for transferring messages between processes. Despite these benefits, messaging with Mach port objects is the least desirable way to communicate between processes. Mach port messaging relies on knowledge of the kernel interfaces, which may change in a future version of Mac OS X.

All other interprocess communications mechanisms in Mac OS X are implemented using Mach ports at some level. As a result, low-level technologies such as sockets, ports, and streams all offer efficient and reliable ways to communicate with other processes. The only time you might consider using Mach ports directly is if you are writing software that runs in the kernel.

User Experience

One reason users choose the Macintosh over other platforms is that it provides a compelling user experience. An important part of this user experience is how third-party applications support the features users need or have come to expect. This section outlines some of the features that users look for and the technologies you can use to support them.

Aqua

Aqua defines the appearance and overall behavior of Mac OS X applications. Aqua applications incorporate color, depth, translucence, and water-like textures into a visually appealing interface. The behavior of an Aqua application is consistent, providing users with familiar paradigms and expected responses to user-initiated actions.

Applications written using modern Mac OS X interfaces (such as those provided by Carbon and Cocoa) get much of the Aqua appearance automatically. However, there is more to Aqua than that. Interface designers must still follow the Aqua guidelines to position windows and controls in

appropriate places. Designers must take into account features such as text, keyboard, and mouse usage and make sure their designs work appropriately for Aqua. The implementers of an interface must then write code to provide the user with appropriate feedback and to convey what is happening in an informative way.

Apple provides the Interface Builder application to assist developers with the proper layout of interfaces. However, you should also be sure to read *Apple Human Interface Guidelines*, which provides invaluable advice on how to create Aqua-compliant applications and on the best Mac OS X interface technologies to use.

Accessibility

Millions of people have some type of disability or special need. Federal regulations in the United States stipulate that computers used in government or educational settings must provide reasonable access for people with disabilities. Mac OS X includes built-in functionality to accommodate users with special needs. It also provides software developers with the functions they need to support accessibility in their own applications.

Applications that use Cocoa or modern Carbon interfaces receive significant support for accessibility automatically. For example, applications get the following support for free:

- Zoom features let users increase the size of onscreen elements.
- Sticky keys let users press keys sequentially instead of simultaneously for keyboard shortcuts.
- Mouse keys let users control the mouse with the numeric keypad.
- Full keyboard access mode lets users complete any action using the keyboard instead of the mouse.
- Speech recognition lets users speak commands rather than type them.
- Text-to-speech reads text to users with visual disabilities.
- VoiceOver provides spoken user interface features to assist visually impaired users.

If your application is designed to work with assistive devices (such as screen readers), you may need to provide additional support. Both Cocoa and Carbon integrate support for accessibility protocols in their frameworks; however, there may still be times when you need to provide additional descriptions or want to change descriptions associated with your windows and controls. In those situations, you can use the appropriate accessibility interfaces to change the settings.

For guidelines on how to support accessibility technologies, see *Apple Human Interface Guidelines*. For more information about the technologies for supporting accessibility, see Getting Started With Accessibility.

AppleScript

Mac OS X employs AppleScript as the primary language for making applications scriptable. AppleScript is supported in all application environments as well as in the Classic compatibility environment. Thus, users can write scripts that link together the services of multiple scriptable applications across different environments.

When designing new applications, you should consider AppleScript support early in the process. The key to good AppleScript design is choosing an appropriate data model for your application. The design must not only serve the purposes of your application but should also make it easy for AppleScript implementers to manipulate your content. Once you settle on a model, you can implement the Apple event code needed to support scripting.

For information about AppleScript in Mac OS X, go to http://www.apple.com/applescript. For developer documentation explaining how to support AppleScript in your programs, see AppleScript Documentation.

Bundles and Packages

A feature integral to Mac OS X software distribution is the bundle mechanism. Bundles encapsulate related resources in a hierarchical file structure but present those resources to the user as a single entity. Programmatic interfaces make it easy to find resources inside a bundle. These same interfaces form a significant part of the Mac OS X internationalization strategy.

Applications and frameworks are only two examples of bundles in Mac OS X. Plug-ins, screen savers, and preference panes are all implemented using the bundle mechanism. Developers can also use bundles for their document types to make it easier to store complex data.

Packages are another technology, similar to bundles, that make distributing software easier. A package—also referred to as an installation package—is a directory that contains files and directories in well-defined locations. The Finder displays packages as files. Double-clicking a package launches the Installer application, which then installs the contents of the package on the user's system.

For an overview of bundles and how they are constructed, see *Bundle Programming Guide*. For information on how to package your software for distribution, see *Software Delivery Guide*.

Fast User Switching

Introduced in Mac OS X version 10.3, fast user switching lets multiple users share physical access to a single computer without logging out. Only one user at a time can access the computer using the keyboard, mouse, and display; however, one user's session can continue to run while another user accesses the computer. The users can then trade access to the computer and toggle sessions back and forth without disturbing each other's work.

When fast user switching is enabled, an application must be careful not to do anything that might affect it adversely if it is running in more than one user session. Shared resources become a particular issue unless they are associated with a particular user session. As you design your application, make sure that any shared resources you use are protected appropriately. For more information on how to do this, see *Multiple User Environments*.

Internationalization and Localization

Localizing your application is necessary for success in many foreign markets. Users in other countries are much more likely to buy your software if the text and graphics reflect their own language and culture. Before you can localize an application, though, you must design it in a way that supports localization, a process called internationalization. Properly internationalizing an application makes it easier for the localization team to replace language-specific or cultural-specific content.

Internationalizing an application involves the following steps:

- Use Unicode strings for storing user-visible text.
- Extract user-visible text into "strings" resource files.
- Use nib files to store window and control layouts whenever possible.
- Use international or culture-neutral icons and graphics whenever possible.
- Use Cocoa or ATSUI to handle text layout.
- Support localized file-system names.
- Use formatter objects in Core Foundation and Cocoa to format numbers, currencies, dates, and times based on the current locale.

For details on how to support localized versions of your software, see *Internationalization Programming Topics*. For information on Core Foundation formatters, see *Data Formatting Guide for Core Foundation*.

Software Configuration

Mac OS X programs commonly use property list files (also known as plist files) to store configuration data. A property list is a text file that uses XML to manage information stored as key-value pairs. Property list files are used for several different types of configuration. For example, applications use a special type of property list file, called an information property list (`Info.plist`) file, to communicate key attributes of the application to the system. Such a file includes the application's name, unique identification string, and version information. Applications also use property list files to store user preferences or other custom configuration data.

There are several tools for creating and modifying property list files. Xcode uses project data to generate a property list file automatically with your bundle. You can also use a text editor or the Property List Editor application that comes with the Xcode Tools. (The Property List Editor is the preferred way to edit property list files because it ensures that the XML structure is valid.) You can also read and write property list files programmatically using facilities found in both Core Foundation and Cocoa. See "XML Parsing" (page 74) for more information.

For more information on managing configuration data, see *Runtime Configuration Guidelines*.

Spotlight

Introduced in Mac OS X version 10.4, Spotlight provides advanced search capabilities for applications. The Spotlight server gathers metadata from documents and other relevant user files and incorporates that metadata into a searchable index. The Finder uses this metadata to provide users with more relevant information about their files. For example, in addition to listing the name of a JPEG file, the Finder can also list its width and height in pixels.

Application developers use Spotlight in two different ways. First, you can search for file attributes and content using the Spotlight search API. Second, if your application defines its own custom file formats, then you should provide a Spotlight importer plug-in to parse files and gather their metadata. In addition to writing an importer, your application should also add metadata information to any new files it creates.

Note: You should not use Spotlight for indexing and searching the general content of a file. Spotlight is intended for searching only the meta information associated with files. To search the actual contents of a file, use the Search Kit API. For more information, see "Search Kit" (page 71).

For more information on using Spotlight in your applications, see *Spotlight Overview*.

System Applications

Mac OS X provides many applications to help both developers and users implement their projects. A default Mac OS X installation includes an `Applications` directory containing many user and administrative tools that you can use in your development. In addition, there are two special applications that are relevant to running programs: the Finder and the Dock.

The Finder

The Finder has many functions in the operating system:

- It is the primary file browser. As such, it is the first tool users see, and one they use frequently to find applications and other files.
- It provides a powerful search tool for finding files not easily found by browsing.
- It provides a way to access servers and other remote volumes, including a user's iDisk.
- It determines the application in which to open a document when a user double-clicks a document icon.
- It allows users to create file archives.
- It provides previews of images, movies, and sounds in its preview pane.
- It lets users burn content onto CDs and DVDs.
- It provides an AppleScript interface for manipulating files and the Finder user interface.

Keep the Finder in mind as you design your application's interface. Understand that any new behaviors you introduce should follow patterns users have grown accustomed to in their use of the Finder. Although some of the functionality of the Finder, like file browsing, is replicated through the Carbon and Cocoa frameworks, the Finder may be where users feel most comfortable performing certain functions. Your application should interact with the Finder gracefully and should communicate changes to the Finder where appropriate. For example, you might want to embed content by allowing users to drag files from the Finder into a document window of your application.

Another way your application interacts with the Finder is through configuration data. The information property list of your bundled application communicates significant information about the application to the Finder. Information about your application's identity and the types of documents it supports are all part of the information property list file.

For information about the Finder and its relationship to the file system, see *File System Overview*. For information about how to configure your application, see *Runtime Configuration Guidelines*.

The Dock

Designed to help prevent onscreen clutter and aid in organizing work, the always available Dock displays an icon for each open application and minimized document. It also contains icons for commonly used applications and for the Trash. In addition, applications can use the Dock to convey information about the application and its current state.

For guidelines on how to work with the Dock within your program, see *Apple Human Interface Guidelines*. For information on how to manipulate Dock tiles in a Carbon application, see *Dock Tile Programming Guide* and *Dock Manager Reference*. To manipulate Dock tiles from a Cocoa application, use the methods of the NSApplication and NSWindow classes.

Application-Level Technologies

This chapter summarizes the application-level technologies that are most relevant to developers—that is, that have programmatic interfaces or have an impact on how you write software. It does not describe user-level technologies, such as Exposé, unless there is some aspect of the technology that allows developer involvement.

If you are new to developing Mac OS X software, you should read through this chapter at least once to understand the available technologies and how you might use them in your software. Even experienced developers should revisit this chapter periodically to remind themselves of the available technologies.

Mac OS X includes several technologies that make developing applications easier. These technologies range from utilities for managing your internal data structures to high-level frameworks for burning CDs and DVDs. Although you are not required to use these technologies, their use is highly encouraged.

Address Book

Mac OS X version 10.2 introduced a centralized database for sharing information about contacts and groups. The database contains information such as user names, street addresses, email addresses, phone numbers, and distribution lists. Applications that support this type of information can use this data as is or extend it to include application-specific information.

The Address Book framework (`AddressBook.framework`) provides a way to access user records and create new ones. Applications that support this framework gain the ability to share user records with other applications, such as the Address Book application and the Apple Mail program. The framework also supports the concept of a "Me" record, which contains information about the currently logged-in user. You can use this record to supply the information automatically; for example, a web browser might use it to populate a web form with the user's address and phone number.

For more information about this technology, see *Address Book Programming Guide*.

Bonjour

Introduced in Mac OS X version 10.2, Bonjour is Apple's implementation of the zero-configuration networking architecture, a powerful system for publishing and discovering services over an IP network. It is relevant to both software and hardware developers.

Incorporating Bonjour support into your software offers a significant improvement to the overall user experience. Rather than prompt the user for the exact name and address of a network device, you can use Bonjour to obtain a list of available devices and let the user choose from that list. For example, you could use it to look for available printing services, which would include any printers or software-based print services, such as a service to create PDF files from print jobs.

Developers of network-based hardware devices are strongly encouraged to support Bonjour. Bonjour alleviates the need for complicated setup instructions for network-based devices such as printers, scanners, RAID servers, and wireless routers. When plugged in, these devices automatically publish the services they offer to clients on the network.

For information on how to incorporate Bonjour services into a Cocoa application, see *Bonjour Overview*. Bonjour for non-Cocoa applications is described in *DNS Service Discovery Programming Guide*.

Core Data

Introduced in Mac OS X version 10.4, the Core Data framework (`CoreData.framework`) is a new technology for managing the model data of a Model-View-Controller application. Core Data is intended for use in Cocoa applications where the data model is already highly structured. Instead of defining data structures programmatically, you use the graphical tools in Xcode to build a schema representing your data model. At runtime, instances of your data-model entities are created, managed, and made available through the Core Data framework.

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

- Storage of object data in mediums ranging from an XML file to a SQLite database
- Management of undo/redo beyond basic text editing
- Support for validation of property values
- Support for propagating changes and ensuring that the relationships between objects remain consistent
- Grouping, filtering, and organizing data in memory and transferring those changes to the user interface through Cocoa bindings

If you are starting to develop a new application, or are planning a significant update to an existing application, you should consider using Core Data. For more information about Core Data, including how to use it in your applications, see *Core Data Programming Guide*.

Core Foundation

The Core Foundation framework (`CoreFoundation.framework`) is a set of C-based interfaces that provide basic data management features for Mac OS X programs. Among the data types you can manipulate with Core Foundation are the following:

- Collections
- Bundles and plug-ins
- Strings
- Raw data blocks
- Dates and times
- Preferences
- Streams
- URLs
- XML data
- Locale information
- Run loops
- Ports and sockets

Although it is C-based, the design of the Core Foundation interfaces is more object-oriented than C. As a result, the opaque types you create with Core Foundation interfaces operate seamlessly with the Cocoa Foundation interfaces. Core Foundation is used extensively in Mac OS X to represent fundamental types of data, and its use in Carbon and other non-Cocoa applications is highly recommended. (For Cocoa applications, use the Cocoa Foundation framework instead.)

For an overview of Core Foundation, see *Design Concepts*. For additional conceptual and reference material, see the categories of Core Foundation Documentation.

Disc Recording

Introduced in Mac OS X version 10.2, the Disc Recording framework (`DiscRecording.framework`) gives applications the ability to burn and erase CDs and DVDs. This framework was built to satisfy the simple needs of a general application, making it easy to add basic audio and data burning capabilities. At the same time, the framework is flexible enough to support professional CD and DVD mastering applications.

The Disc Recording framework minimizes the amount of work your application must perform to burn optical media. Your application is responsible for specifying the content to be burned but the framework takes over the process of buffering the data, generating the proper file format information, and communicating everything to the burner. In addition, the Disc Recording UI framework (`DiscRecordingUI.framework`) provides a complete, standard set of windows for gathering information from the user and displaying the progress of the burn operation.

The Disc Recording framework supports applications built using Carbon and Cocoa. The Disc Recording UI framework currently provides user interface elements for Cocoa applications only.

HeaderDoc documentation for the Disc Recording framework is provided in Music & Audio Documentation.

Help

Although some applications are extremely simple to use, most require some documentation. Help tags (also called tooltips) and documentation are the best ways to provide users with immediate answers to questions. Help tags provide descriptive information about your user interface quickly and unobtrusively. Documentation provides more detailed solutions to problems, including conceptual material and task-based examples. Both of these elements help the user understand your user interface better and should be a part of every application.

For information on adding help to a Cocoa application, see *Online Help*. For information on adding help to a Carbon application, see *Providing User Assistance With Apple Help (Revision; Preliminary)*.

Human Interface Toolbox

Introduced in Mac OS X version 10.2, the Human Interface Toolbox (HIToolbox) provides HIObject, HIView, and several other objects for organizing windows, controls, and menus in Carbon applications. The HIObject model builds on Core Foundation data types to bring a modern, object-oriented approach to the HIToolbox. Although the model is object-oriented, access to the objects is handled by a set of C interfaces. Using the HIToolbox interfaces is recommended for the development of new Carbon applications. Some benefits of this technology include the following:

- Drawing is handled natively using Quartz.
- A simplified, modern coordinate system is used that is not bounded by the 16-bit space of QuickDraw.
- Layering of views is handled automatically.
- Views can be attached and detached from windows.
- Views can be hidden temporarily.

For reference material and an overview of HIObject, HIView, and other HIToolbox objects, see the documents in Carbon Human Interface Toolbox Documentation.

iChat Presence

Introduced in Mac OS X version 10.4, the Instant Message framework (`InstantMessage.framework`) supports the detection and display of a user's online presence in applications other than iChat. Using this framework, you can find out the current status of a user connected to an instant messaging service. You can then obtain the user's custom icon, status message, or a URL to an image that indicates the

user's status. You can then display this information along with other user information in your program. For example, Mail uses the framework to determine if an email is from a user who is currently online; it the person is available, it then displays an appropriate icon next to that person's name.

For more information, see *iChat Instant Message Framework Reference*.

Image Capture

The Image Capture framework (`ICADevices.framework`) is a high-level framework for capturing image data from scanners and digital cameras. The interfaces of the framework are device-independent, so you can use it to gather data from any devices connected to the system. You can get a list of devices, retrieve information about a specific device or image, and retrieve the image data itself.

Ink

The Ink feature of Mac OS X provides handwriting recognition for applications that support the Carbon and Cocoa text systems. However, the automatic support provided by these text systems is limited to basic recognition. The Ink framework offers several additional features that you can incorporate into your applications, including the following:

- Enable or disable handwriting recognition programmatically
- Access Ink data directly
- Support either deferred recognition or recognition on demand
- Support the direct manipulation of text by means of gestures

The Ink framework is not limited to developers of end-user applications. Hardware developers can also use it to implement a handwriting recognition solution for a new input device. You might also use the Ink framework to implement your own correction model to provide users with a list of alternate interpretations for handwriting data.

Ink is included as a subframework of `Carbon.framework`. For more information on using Ink in Carbon and Cocoa applications, see *Using Ink Services in Your Application*.

Keychain Services

Keychain Services provides a secure way to store passwords, keys, certificates, and other sensitive information for a user. Users often have to manage multiple user IDs and passwords to access various login accounts, servers, secure websites, instant messaging services, and so on. A keychain is an encrypted container that holds passwords for multiple applications and secure services. Access to the keychain is provided through a single master password. Once the keychain is unlocked, Keychain Services–aware applications can access authorized information without bothering the user.

Users with multiple accounts tend to manage those accounts in the following ways:

- They create a simple, easily remembered password.
- They repeatedly use the same password.
- They write the password down where it can easily be found.

If your application handles passwords or sensitive information, you should add support for Keychain Services into your application. For more information on this technology, see *Keychain Services Programming Guide*.

Launch Services

Launch Services provides a programmatic way for you to open applications, documents, URLs, or files with a given MIME type in a way similar to the Finder or the Dock. It makes it easy to open documents in the user's preferred application or open URLs in the user's favorite web browser. The Launch Services framework also provides interfaces for programmatically registering the document types your application supports.

For information on how to use Launch Services, see *Launch Services Concepts and Tasks*.

Open Directory

Open Directory is a directory services architecture that provides a centralized way to retrieve information stored in local or network databases. Directory services typically provide access to collected information about users, groups, computers, printers, and other information that exists in a networked environment (although they can also store information about the local system). You use Open Directory in your programs to retrieve information from these local or network databases. For example, if you're writing an email program, you can use Open Directory to connect to a corporate LDAP server and retrieve the list of individual and group email addresses for the company.

Open Directory uses a plug-in architecture to support a variety of retrieval protocols. Mac OS X provides plug-ins to support LDAPv2, LDAPv3, NetInfo, AppleTalk, SLP, SMB, DNS, Microsoft Active Directory, and Bonjour protocols, among others. You can also write your own plug-ins to support additional protocols.

For more information on this technology, see *Open Directory Programming Guide*. For information on how to write Open Directory plug-ins, see *Open Directory Plug-in Programming Guide*.

PDF Kit

Introduced in Mac OS X version 10.4, PDF Kit is a Cocoa framework for managing and displaying PDF content directly from your application's windows and dialogs. Using the classes of the PDF Kit, you can embed a PDFView in your window and give it a PDF file to display. The PDFView class handles the rendering of the PDF content, handles copy-and-paste operations, and provides controls

for navigating and setting the zoom level. Other classes let you get the number of pages in a PDF file, find text, manage selections, add annotations, and specify the behavior of some graphical elements, among other actions. Users can also copy selected text in a PDFView to the pasteboard.

Note: Although it is written in Objective-C, you can use the classes of the PDF Kit in both Carbon and Cocoa applications. For information on how to do this, see *Carbon-Cocoa Integration Guide*.

If you need to display PDF data directly from your application, the PDF Kit is highly recommended. It hides many of the intricacies of the Adobe PDF specification and provides standard PDF viewing controls automatically. The PDF Kit is part of the Quartz framework (`Quartz.framework`). For more information, see *PDF Kit Reference Collection*.

QuickTime Kit

Introduced in Mac OS X version 10.4, the QuickTime Kit (`QTKit.framework`), is an Objective-C framework for manipulating QuickTime-based media. This framework lets you incorporate movie playback, movie editing, export to standard media formats, and other QuickTime behaviors easily into your applications. The classes in this framework open up a tremendous amount of QuickTime behavior to both Carbon and Cocoa developers. Instead of learning how to use the more than 2500 functions in QuickTime, you can now use a handful of classes to implement the features you need.

Note: The QuickTime Kit framework supersedes the NSMovie and NSMovieView classes available in Cocoa. If your code uses these older classes, you should consider changing your code to use the QuickTime Kit.

For an introduction and tutorial on how to use the QuickTime Kit, see *QuickTime Kit Programming Guide*. For reference information about the QuickTime Kit classes, see *QuickTime Kit Framework Reference*.

Search Kit

Introduced in Mac OS X version 10.3, the Search Kit framework lets you search, summarize, and retrieve documents written in most human languages. You can incorporate these capabilities into your application to support fast searching of content managed by your application.

The Search Kit framework is part of the Core Services umbrella framework. The technology is derived from the Apple Information Access Toolkit, which is often referred to by its code name V-Twin. Many system applications, including Spotlight, Finder, Address Book, Apple Help, and Mail use this framework to implement searching.

Search Kit is an evolving technology and as such continues to improve in speed and features. For detailed information about the available features, see *Search Kit Reference*.

Security Services

Mac OS X security is built using several open source technologies, including BSD, Common Data Security Architecture (CDSA), and Kerberos. Mac OS X builds on these basic technologies by implementing a layer of high-level services to simplify your security solutions. These high-level services provide a convenient abstraction and make it possible for Apple and third parties to implement new security features without breaking your code. They also make it possible for Apple to combine security technologies in unique ways; for example, Keychain Services provides encrypted data storage with authenticated access using several CDSA technologies.

Mac OS X provides high-level interfaces for the following features:

- User authentication
- Certificate, key, and trust services
- Authorization services
- Secure transport
- Keychain Services

Mac OS X supports many network-based security standards, including SFTP, S/MIME, and SSH. For a complete list of network protocols, see "Standard Network Protocols" (page 41).

For more information about the security architecture and security-related technologies of Mac OS X, see *Security Overview*. For additional information about CDSA, see the following page of the Open Group's website: http://www.opengroup.org/security/cdsa.htm.

Speech Technologies

Mac OS X contains speech technologies that recognize and speak U.S. English. These technologies provide benefits for users and present the possibility of a new paradigm for human-computer interaction.

Speech recognition is the ability for the computer to recognize and respond to a person's speech. Using speech recognition, users can accomplish tasks comprising multiple steps with one spoken command. Because users control the computer by voice, speech-recognition technology is very important for people with special needs. You can take advantage of the speech engine and API included with Mac OS X to incorporate speech recognition into your applications.

Speech synthesis, also called text-to-speech (TTS), converts text into audible speech. TTS provides a way to deliver information to users without forcing them to shift attention from their current task. For example, the computer could deliver messages such as "Your download is complete" and "You have email from your boss; would you like to read it now?" in the background while you work. TTS is crucial for users with vision or attention disabilities. As with speech recognition, Mac OS X TTS provides an API and several user interface features to help you incorporate speech synthesis into your applications. You can also use speech synthesis to replace digital audio files of spoken text. Eliminating these files can reduce the overall size of your software bundle.

For more information, see User Experience Speech Technologies Documentation.

SQLite

Introduced in Mac OS X version 10.4, the SQLite library lets you embed a SQL database engine into your applications. Programs that link with the SQLite library can access SQL databases without running a separate RDBMS process. 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 SQLite library is located at `/usr/lib/libsqlite3.dylib` and the `sqlite3.h` header file is in `/usr/include`. A command-line interface (`sqlite3`) is also available for communicating with SQLite databases using scripts. For details on how to use this command-line interface, see the man page for `sqlite3`. You can access this page from the command line or in *Mac OS X Man Pages*.

For more information about using SQLite, go to http://www.sqlite.org.

Sync Services

Introduced in Mac OS X version 10.4, Sync Services extends data synchronization capabilities to all Mac OS X applications. Third-party applications can use Sync Services to synchronize data with system databases, such as those provided by Address Book and iCal. They can also synchronize custom data with other applications and across multiple computers through the user's .Mac account.

Note: With Sync Services, applications can now directly initiate the synchronization process. The iSync application still exists but is now used to initiate the synchronization process for specific hardware devices.

The Sync Services framework (`SyncServices.framework`) provides an Objective-C interface but can be used by both Carbon and Cocoa applications. Applications with structured data can use this framework to initiate sync sessions and to push and pull records from the central "truth" database, which the sync engine uses to maintain the master copy of the synchronized records. The system provides predefined schemas for contacts, calendars, and bookmarks in the `/System/Library/SyncServices/Schemas` directory. You can also distribute custom schemas for your own data types and register them with Sync Services.

For more information about using Sync Services in your application, see *Sync Services Programming Guide*. For reference information, see *Sync Services Reference*.

Web Kit

Introduced in Mac OS X version 10.3, the Web Kit framework provides an engine for displaying HTML-based content. The Web Kit framework is an umbrella framework containing two subframeworks: Web Core and JavaScript Core. The Web Core framework is based on the kHTML rendering engine, an open source engine for parsing and displaying HTML content. The JavaScript Core framework is based on the KJS open source library for parsing and executing JavaScript code.

Starting with Mac OS X version 10.4, Web Kit also lets you create text views containing editable HTML. The editing support is equivalent to the support available in Cocoa for editing RTF-based content. With this support, you can replace text and manipulate the document text and attributes, including CSS properties. Although it offers many features, the Web Kit editing support is not intended to provide a full-featured editing facility like you might find in professional HTML editing applications. Instead, it is aimed at developers who need to display HTML and handle the basic editing of HTML content.

Also introduced in Mac OS X version 10.4, Web Kit includes support for creating and editing content at the DOM level of an HTML document. You can use this support to navigate DOM nodes and manipulate those nodes and their attributes. You can also use the framework to extract DOM information. 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 on how to use Web Kit from both Carbon and Cocoa applications, see *Web Kit Objective-C Programming Guide*. For information on the classes and protocols in the Web Kit framework, see *Web Kit Objective-C Framework Reference*.

Web Service Access

Many businesses provide web services for retrieving data from their websites. The available services cover a wide range of information and include things such as financial data and movie listings. Mac OS X has included support for calling web-based services using Apple events since version 10.1. However, starting with version 10.2, the Web Services Core framework (part of the Core Services umbrella framework) provides support for the invocation of web services using CFNetwork.

For a description of web services and information on how to use the Web Services Core framework, see *Web Services Core Programming Guide*.

XML Parsing

The Darwin layer of Mac OS X version 10.3 and later includes the `libXML2` library for parsing XML data. This is an open source library that you can use to parse or write arbitrary XML data quickly. The headers for this library are located in the `/usr/include/libxml2` directory.

Several other XML parsing technologies are also included in Mac OS X. For arbitrary XML data, Core Foundation provides a set of functions for parsing the XML content from Carbon or other C-based applications. Cocoa provides several classes to implement XML parsing. If you need to read or write a property list file, you can use either the Core Foundation CFPropertyList functions or the Cocoa NSDictionary object to build a set of collection objects with the XML data.

For information on Core Foundation support for XML parsing, see the documents in Core Foundation Data Management Documentation. For information on parsing XML from a Cocoa application, see *Tree-Based XML Programming Guide for Cocoa*.

Choosing Technologies to Match Your Design Goals

Mac OS X has many layers of technology. Before choosing a specific technology to implement a solution, think about the intended role for that technology. Does that technology meet the needs of your design goals? In some cases, Mac OS X might offer several technologies that you could use but some choices might be more appropriate. For example, you might want to forego a higher-level API in order to get the maximum possible performance offered by a lower-level API.

The choice of technologies depends on your overall goals. If performance is a key goal, then you would choose technologies that offer the best performance but which perhaps require more custom code to implement.

The following sections list some of the high-level goals you should strive for in your Mac OS X software. Along with each goal are a list of some technologies that can help you achieve that goal. These lists are not exhaustive but provide you with ideas you might not have considered otherwise. For specific design tips related to these goals, see *Apple Human Interface Guidelines*.

High Performance

Performance is the perceived measure of how fast or efficient your software is, and it is critical to the success of all software. If your software seems slow, users may be less inclined to buy it. Even software that uses the most optimal algorithms may seem slow if it spends more time processing data than responding to the user.

Developers who have experience programming on other platforms (including Mac OS 9) should take the time to learn about the factors that influence performance on Mac OS X. Understanding these factors can help you make better choices in your design and implementation. For information about performance factors and links to performance-related documentation, see *Performance Overview*.

Table 5-1 lists several Mac OS X technologies that you can use to improve the performance of your software.

Table 5-1 Technologies for improving performance

Technology	Description
Velocity Engine	Velocity Engine is a hardware-based vector processing unit that lets you perform the same operation on multiple scalar or floating-point values simultaneously. You can use the Accelerate Framework and vDSP library to perform accelerated math calculations, DSP, and image processing.
Threads	Mac OS X includes several threading packages to meet your needs. For information on these packages and how to use them, see *Multithreading Programming Topics*.
Mach-O executable format	Mach-O is the native executable format of Mac OS X and provides the best performance when you are calling system libraries. See *Mac OS X ABI Mach-O File Format Reference*.
Apple performance tools	Apple provides a suite of performance tools for measuring many aspects of your software. Use these tools to identify hot spots and gather performance metrics that can help identify potential problems. For a list of available tools, see *Performance Overview*.

Mac OS X supports both modern and legacy APIs. Most of the legacy APIs derive from the assorted managers that were part of the original Macintosh Toolbox and are now a part of Carbon. While many of these APIs still work in Mac OS X, they are not as efficient as APIs created specifically for Mac OS X. In fact, many APIs that provided the best performance in Mac OS 9 now provide the worst performance in Mac OS X because of fundamental differences in the two architectures.

Note: For specific information about legacy Carbon managers and the recommended replacements for them, see "Carbon Considerations" (page 83).

As Mac OS X evolves, the list of APIs and technologies it encompasses may change to meet the needs of developers. As part of this evolution, less efficient interfaces may be deprecated in favor of newer ones. Apple makes these changes only when deemed absolutely necessary and uses the availability macros (defined in `/usr/include/AvailabilityMacros.h`) to help you find deprecated interfaces. When you compile your code, deprecated interfaces trigger the generation of compiler warnings. Use these warnings to find deprecated interfaces, and then check the corresponding reference documentation or header files to see if there are recommended replacements.

Easy to Use

An easy-to-use program offers a compelling, intuitive experience for the user. It offers elegant solutions to complex problems and has a well thought out interface that uses familiar paradigms. It is easy to install and configure because it makes intelligent choices for the user, but it also gives the user the option to override those choices when needed. It presents the user with tools that are relevant in the current context, eliminating or disabling irrelevant tools. It also warns the user against performing dangerous actions and provides ways to undo those actions if taken.

For information on designing an easy-to-use interface, see *Apple Human Interface Guidelines*.

Table 5-2 lists several Mac OS X technologies that you can use to make your software easier to use.

Table 5-2 Technologies for achieving ease of use

Technology	Description
Accessibility technologies	The Accessibility interfaces for Carbon and Cocoa make it easier for people with disabilities to use your software. See Accessibility Documentation for a list of documents related to improving software access for people with disabilities.
Address Book framework	If your program uses contact information, the Address Book framework is a must. Supporting this framework means that users don't have to reenter contact information in your program; they can use data they've already entered in Address Book–aware programs. See *Address Book Programming Guide* for more information.
AppleScript	AppleScript makes it possible for users to automate complex workflows quickly. It also gives users a powerful tool for controlling your application. See *AppleScript Overview* for an overview of this technology.
Aqua	If your program has a visual interface, it should adhere to the human interface guidelines for Aqua, which include tips for how to lay out your interface and manage its complexity. *Apple Human Interface Guidelines* explains these guidelines and is an invaluable resource.
Disc Recording framework	The Disc Recording framework provides a consistent interface for burning content onto CD and DVD discs. It posts appropriate dialogs and manages the burn process for you. Reference documentation for this framework is available in Carbon File Management Documentation and Cocoa Documentation.
DVD Playback framework	The DVD Playback framework offers a standard interface for accessing and playing the contents of a DVD.
Keychain Services	Keychains provide users with secure access to passwords, certificates, and other secret information. Adding support for Keychain Services in your program can reduce the number of times you need to prompt the user for passwords and other secure information. For more information, see *Keychain Services Programming Guide*.
Printing	The standard printing dialogs provide a consistent printing interface and should always be used. These dialogs also provide automatic support for print preview, fax, and save-as-PDF features. For more information about printing, see *Mac OS X Printing System Overview*.
Bonjour	Bonjour simplifies the process of configuring and detecting network services. Your program can vend network services or use Bonjour to be a client of an existing network service. See *Bonjour Overview* and *DNS Service Discovery Programming Guide* for information about Bonjour.
Internationalization	Mac OS X provides significant infrastructure for internationalizing software bundles. For guidelines and information about how to internationalize your programs, see *Internationalization Programming Topics*.

Attractive Appearance

One feature that draws users to the Macintosh platform, and to Mac OS X in particular, is the stylish design and attractive appearance of the hardware and software. Although creating attractive hardware and system software is Apple's job, you should take advantage of the strengths of Mac OS X to give your own programs an attractive appearance.

The Finder and other programs that come with Mac OS X use high-resolution, high-quality graphics and icons that include 32-bit color and transparency. You should make sure that your programs also use high-quality graphics both for the sake of appearance and to better convey relevant information to users. For example, the system uses pulsing buttons to identify the most likely choice and transparency effects to add a dimensional quality to windows.

Table 5-3 lists several Mac OS X technologies you can use to ensure that your software has an attractive appearance.

Table 5-3 Technologies for achieving an attractive appearance

Technology	Description
Application Kit	Cocoa includes several classes for rendering simple and complex graphics, including text, and these classes are well suited for application drawing. For more information about drawing in Cocoa, start with *Cocoa Drawing Guide* and then follow up with some of the other documents in Cocoa Graphics & Imaging Documentation.
Aqua	*Apple Human Interface Guidelines* tells you how to create applications that are Aqua-compliant. All Mac OS X applications should follow these guidelines to ensure an attractive look and feel for users.
ATSUI, MLTE	Apple Type Services for Unicode Imaging (ATSUI) and Multilingual Text Engine (MLTE) support high-quality rendering and layout of Unicode text for Carbon applications. These technologies are also built in to the text-handling classes of the Application Kit. See Getting Started With Text & Fonts in Text & Fonts Documentation for more information.
OpenGL	OpenGL is the preferred 3D rendering API for Mac OS X. The Mac OS X implementation of OpenGL is hardware accelerated on many systems and has all of the standard OpenGL support for shading and textures. See *OpenGL Programming Guide for Mac OS X* for an overview of OpenGL and guidelines on how to use it. For information about using OpenGL with Cocoa, see *Cocoa OpenGL* .
Quartz	Quartz is the native (and preferred) 2D rendering API for Mac OS X. It provides primitives for rendering text, images, and vector shapes and includes integrated color management and transparency support. See *Quartz 2D Programming Guide* for details.

Reliability

A reliable program is one that earns the user's trust. Such a program presents information to the user in an expected and desired way. A reliable program maintains the integrity of the user's data and does everything possible to prevent data loss or corruption. It also has a certain amount of maturity to it and can handle complex situations without crashing.

Reliability is important in all areas of software design, but especially in areas where a program may be running for an extended period of time. For example, scientific programs often perform calculations on large data sets and can take a long time to complete. If such a program were to crash during a long calculation, the scientist could lose days or weeks worth of work.

As you start planning a new project, put some thought into what existing technologies you can leverage from both Mac OS X and the open-source community. For example, if your application displays HTML documents, it doesn't make sense to write your own HTML parsing engine when you can use the Web Kit framework instead.

By using existing technologies, you reduce your development time by reducing the amount of new code you have to write and test. You also improve the reliability of your software by using code that has already been designed and tested to do what you need.

Using existing technologies has other benefits as well. For many technologies, you may also be able to incorporate future updates and bug fixes for free. Apple provides periodic updates for many of its shipping frameworks and libraries, either through software updates or through new versions of Mac OS X. If your application links to those frameworks, it receives the benefit of those updates automatically.

All of the technologies of Mac OS X offer a high degree of reliability. However, Table 5-4 lists some specific technologies that improve reliability by reducing the amount of complex code you have to write from scratch.

Table 5-4 Technologies for achieving reliability

Technology	Description
ATSUI/MLTE/Cocoa text system	ATSUI, MLTE, and the Cocoa text system let you render text in multiple languages, fonts, and script systems in the same document, which is normally a very complex operation. Using these technologies is much simpler and more reliable than attempting to do this rendering on your own. For more information, see *ATSUI Reference*, *Multilingual Text Engine Reference*, or *Text System Overview*.
Authorization Services	Authorization Services provides a way to ensure that only authorized operations take place. Preventing unauthorized access helps protect your program as well as the rest of the system. See *Performing Privileged Operations With Authorization Services* for more information.
ColorSync	ColorSync helps ensure color consistency among different devices such as scanners, printers, and the user's display. See *Managing Colors With ColorSync in Mac OS 9* for an overview of the ColorSync technology. See *ColorSync Manager Reference* for information about the ColorSync interfaces.

Technology	Description
Core Foundation	Core Foundation supports basic data types and eliminates the need for you to implement string and collection data types, among others. Both Carbon and Cocoa support the Core Foundation data types, which makes it easier for you to integrate them into your own data structures. See Getting Started With Core Foundation for more information.
Web Kit	The Web Kit provides a reliable, standards-based mechanism for rendering HTML content (including JavaScript code) in your application.

Adaptability

An adaptable program is one that adjusts appropriately to its surroundings; that is, it does not stop working when the current conditions change. If a network connection goes down, an adaptable program lets the user continue to work offline. Similarly, if certain resources are locked or become unavailable, an adaptable program finds other ways to meet the user's request.

One of the strengths of Mac OS X is its ability to adapt to configuration changes quickly and easily. For example, if the user changes a computer's network configuration from the system preferences, the changes are automatically picked up by applications such as Safari and Mail, which use CFNetwork to handle network configuration changes automatically.

Table 5-5 lists some Mac OS X technologies that you can use to improve the overall adaptability of your software.

Table 5-5 Technologies for achieving adaptability

Technology	Description
Core Foundation	Core Foundation provides services for managing date, time, and number formats based on any locale. See Core Foundation Documentation for specific reference documents.
I/O Kit	Before writing a custom driver, see if the I/O Kit provides a driver that meets your needs. *I/O Kit Fundamentals* provides an overview of the available I/O Kit driver families.
Quartz Services	Quartz Services provides access to screen information and provides notifications when screen information changes. See *Quartz Display Services Reference* for more information.
Bonjour	Bonjour simplifies the process of configuring and detecting network services. You can use it to gather information about available network services. See *Bonjour Overview* and *DNS Service Discovery Programming Guide* for information about Bonjour.
System Configuration	The System Configuration framework provides information about availability of network entities. See *System Configuration Framework Reference* and *System Configuration Programming Guidelines* for more information.

Interoperability

Interoperability refers to a program's ability to communicate across environments. This communication can occur at either the user or the program level and can involve processes on the current computer or on remote computers. At the program level, an interoperable program supports ways to move data back and forth between itself and other programs. It might therefore support the pasteboard and be able to read file formats from other programs on either the same or a different platform. It also makes sure that the data it creates can be read by other programs on the system.

Users see interoperability in features such as the pasteboard (the Clipboard in the user interface), drag and drop, AppleScript, Bonjour, and services in the Services menu. All of these features provide ways for the user to get data into or out of an application.

Table 5-6 lists some Mac OS X technologies that you can use to improve the interoperability of your software.

Table 5-6 Technologies for achieving interoperability

Technology	Description
AppleScript	AppleScript is a scripting system that gives users direct control over your application as well as parts of Mac OS X. See *AppleScript Overview* for information on supporting AppleScript.
Drag and drop	Although primarily implemented in applications, you can add drag and drop support to any program with a user interface. See *Drag Manager Reference* or *Drag and Drop Programming Topics for Cocoa* for information on how to integrate drag and drop support into your program.
Pasteboard	Both Carbon and Cocoa support cut, copy, and paste operations through the pasteboard. See the `Pasteboard.h` header file in the HIServices framework or *Pasteboard Programming Topics for Cocoa* for information on how to support the pasteboard in your program.
Bonjour	Bonjour lets your program vend or discover network services. See *Bonjour Overview* and *DNS Service Discovery Programming Guide* for information about Bonjour.
Services	Services let the user perform a specific operation in your application using data on the pasteboard. Services use the pasteboard to exchange data but act on that data in a more focused manner than a standard copy-and-paste operation. For example, a service might create a new mail message and paste the data into the message body. See *Setting Up Your Carbon Application to Use the Services Menu* or *System Services* for information on setting up an application to use services.
XML	XML is a structured format that can be used for data interchange. Mac OS X provides extensive support for reading, writing, and parsing XML data. See *XML Programming Topics for Core Foundation* or the documentation for the Foundation class NSXMLParser.

Mobility

Designing for mobility has become increasingly important as laptop usage soars. A program that supports mobility doesn't waste battery power by polling the system or accessing peripherals unnecessarily, nor does it break when the user moves from place to place, changes monitor configurations, puts the computer to sleep, or wakes the computer up.

To support mobility, programs need to be able to adjust to different system configurations, including network configuration changes. Many hardware devices can be plugged in and unplugged while the computer is still running. Mobility-aware programs should respond to these changes gracefully. They should also be sensitive to issues such as power usage. Constantly accessing a hard drive or optical drive can drain the battery of a laptop quickly. Be considerate of mobile users by helping them use their computer longer on a single battery charge.

Table 5-7 lists some Mac OS X technologies that you can use to improve the mobility of your software.

Table 5-7 Technologies for achieving mobility

Technology	Description
CFNetwork	CFNetwork provides a modern interface for accessing network services and handling changes in the network configuration. See *CFNetwork Programming Guide* for an introduction to the CFNetwork API.
Quartz Services	Quartz Services provides access to screen information and provides notifications when screen information changes. See *Quartz Display Services Reference* for information about the API.
Bonjour	Bonjour lets mobile users find services easily or vend their own services for others to use. See *Bonjour Overview* and *DNS Service Discovery Programming Guide* for information about Bonjour.
System Configuration	The System Configuration framework is the foundation for Apple's mobility architecture. You can use its interfaces to get configuration and status information for network entities. It also sends out notifications when the configuration or status changes. See *System Configuration Framework Reference* or *System Configuration Programming Guidelines* for more information.

Porting Tips

Although many applications have been created from scratch for Mac OS X, many more have been ported from existing Windows, UNIX, or Mac OS 9 applications. With the introduction of the G5 processor, some application developers are even taking the step of porting their 32-bit applications to the 64-bit memory space offered by the new architecture.

The Porting Documentation section of the Apple Developer Connection Reference Library contains documents to help you in your porting efforts. The following sections also provide general design guidelines to consider when porting software to Mac OS X.

Carbon Considerations

If you develop your software using Carbon, there are several things you can do to make your programs work better in Mac OS X. The following sections list migration tips and recommendations for technologies you should be using.

Migrating From Mac OS 9

If you were a Mac OS 9 developer, the Carbon interfaces should seem very familiar. However, improvements in Carbon have rendered many older technologies obsolete. The sections that follow list both the required and the recommended replacement technologies you should use instead.

Required Replacement Technologies

The technologies listed in Table 6-1 cannot be used in Carbon. You must use the technology in the "Now use" column instead.

Table 6-1 Required replacements for Carbon

Instead of	Now use
Any device manager	I/O Kit
Apple Guide	Apple Help
AppleTalk Manager	BSD sockets or CFNetwork

Instead of	Now use
Help Manager	Carbon Help Manager
PPC Toolbox	Apple events
Printing Manager	Carbon Printing Manager
QuickDraw 3D	OpenGL
QuickDraw GX	Quartz and Apple Type Services for Unicode Imaging (ATSUI)
Standard File Package	Navigation Services
Vertical Retrace Manager	Time Manager

Recommended Replacement Technologies

The technologies listed in Table 6-2 can still be used in Carbon, but the indicated replacements provide more robust support and are preferred.

Table 6-2 Recommended replacements for Carbon

Instead of	Now use
Display Manager	Quartz Services
Event Manager	Carbon Event Manager
Font Manager	Apple Type Services for Fonts
Internet Config	Launch Services and System Configuration
Open Transport	BSD sockets or CFNetwork
QuickDraw	Quartz 2D
QuickDraw Text	Multilingual Text Engine (MLTE) or Apple Type Services for Unicode Imaging (ATSUI)
Resource Manager	Interface Builder Services
Script Manager	Unicode Utilities
TextEdit	Multilingual Text Engine
URL Access Manager	CFNetwork

Use the Carbon Event Manager

Use of the Carbon Event Manager is strongly recommended for new and existing Carbon applications. The Carbon Event Manager provides a more robust way to handle events than the older Event Manager interfaces. For example, the Carbon Event Manager uses callback routines to notify your application when an event arrives. This mechanism improves performance and offers better mobility support by eliminating the need to poll for events.

For an overview of how to use the Carbon Event Manager, see *Carbon Event Manager Programming Guide*.

Use the HIToolbox

The Human Interface Toolbox is the technology of choice for implementing user interfaces with Carbon. The HIToolbox extends the Macintosh Toolbox and offers an object-oriented approach to organizing the content of your application windows. This new approach to user interface programming is the future direction for Carbon and is where new development and improvements are being made. If you are currently using the Control Manager and Window Manager, you should consider adopting the HIToolbox.

For an overview of HIView and other HIToolbox objects, see the documents in Carbon Human Interface Toolbox Documentation.

Use Nib Files

Nib files, which you create with Interface Builder, are the best way to design your application interface. The design and layout features of Interface Builder will help you create Aqua-compliant windows and menus. Even if you do not plan to load the nib file itself, you can still use the metrics from this file in your application code.

Windows Considerations

If you are a Windows developer porting your application to Mac OS X, be prepared to make some changes to your application as part of your port. Applications in Mac OS X have an appearance and behavior that are different from Windows applications in many respects. Unless you keep these differences in mind during the development cycle, your application may look out of place in Mac OS X.

The following list provides some guidelines related to the more noticeable differences between Mac OS X and Windows applications. This list is not exhaustive but is a good starting point for developers new to Mac OS X. For detailed information on how your application should look and behave in Mac OS X, see *Apple Human Interface Guidelines*. For general porting information, see *Porting to Mac OS X from Windows Win32 API*.

- **Avoid custom controls**. Avoid creating custom controls if Mac OS X already provides equivalent controls for your needs. Custom controls are appropriate only in situations where the control is unique to your needs and not provided by the system. Replacing standard controls can make your interface look out of place and might confuse users.
- **Use a single menu bar**. The Mac OS X menu bar is always at the top of the screen and always contains the commands for the currently active application. You should also pay attention to the layout and placement of menu bar commands, especially commonly used commands such as New, Open, Quit, Copy, Minimize, and Help.
- **Pay attention to keyboard shortcuts**. Mac OS X users are accustomed to specific keyboard shortcuts and use them frequently. Do not simply migrate the shortcuts from your Windows application to your Mac OS X application. Also remember that Mac OS X uses the Command key not the Control key as the main keyboard modifier.
- **Do not use MDI**. The Multiple Document Interface (MDI) convention used in Microsoft Windows directly contradicts Mac OS X design guidelines. Windows in Mac OS X are document-centric and not application-centric. Furthermore, the size of a document window is constrained only by the user's desktop size.
- **Use Aqua**. Aqua gives Mac OS X applications the distinctive appearance and behavior that users expect from the platform. Using nonstandard layouts, conventions, or user interface elements can make your application seem unpolished and unprofessional.
- **Design high-quality icons**. Mac OS X icons are often displayed in sizes varying from 16x16 to 128x128 pixels. These icons are usually created professionally, with millions of colors and photo-realistic qualities. Your application icons should be vibrant and inviting and should immediately convey your application's purpose.
- **Design clear and consistent dialogs**. Use the standard Open, Save, printing, Colors, and Font dialogs in your applications. Make sure alert dialogs follow a consistent format, indicating what happened, why it happened, and what to do about it.
- **Consider toolbars carefully**. Having a large number of buttons, especially in an unmovable toolbar, contributes to visual clutter and should be avoided. When designing toolbars, include icons only for menu commands that are not easily discoverable or that may require multiple clicks to be reached.
- **Use an appropriate layout for your windows**. The Windows user interface relies on a left-biased, more crowded layout, whereas Aqua relies on a center-biased, spacious layout. Follow the Aqua guidelines to create an appealing and uncluttered interface that focuses on the task at hand.
- **Avoid application setup steps**. Whenever possible, Mac OS X applications should be delivered as drag-and-drop packages. If you need to install files in multiple locations, use an installation package to provide a consistent installation experience for the user. If your application requires complex setup procedures in order to run, use a standard Mac OS X assistant.
- **Use filename extensions**. Mac OS X fully supports and uses filename extensions. However, be aware that you may also need to supply file type and creator type codes if you intend for your documents to be viewed in Mac OS 9. For more information about filename extensions, see *File System Overview*.

64-Bit Considerations

The introduction of the PowerPC G5 means that developers can begin writing software to take advantage of the 64-bit architecture provided by this chip. For many developers, however, compiling their code into 64-bit programs may not offer any inherent advantages. Unless your program needs more than 4 GB of addressable memory, supporting 64-bit pointers may only reduce the performance of your application.

When you compile a program for a 64-bit architecture, the compiler doubles the size of all pointer variables. This increased pointer size makes it possible to address more than 4 GB of memory, but it also increases the memory footprint of your application. If your application does not take advantage of the expanded memory limits, it may be better left as a 32-bit program.

Regardless of whether your program is currently 32-bit or 64-bit, there are some guidelines you should follow to make your code more interoperable with other programs. Even if you don't plan to implement 64-bit support soon, you may need to communicate with 64-bit applications. Unless you are explicit about the data you exchange, you may run into problems. The following guidelines are good to observe regardless of your 64-bit plans.

- Avoid casting pointers to anything but a pointer. Casting a pointer to a scalar value has different results for 32-bit and 64-bit programs. These differences could be enough to break your code later or cause problems when your program exchanges data with other programs.
- Be careful not to make assumptions about the size of pointers or other scalar data types. If you want to know the size of a type, always use the `sizeof` (or equivalent) operator.
- If you write integer values to a file, make sure your file format specifies the exact size of the value. For example, rather than assume the generic type `int` is 32 bits, use the more explicit types `SInt32` or `int32_t`, which are guaranteed to be the correct size.
- If you exchange integer data with other applications across a network, make sure you specify the exact size of the integer.

For more information about writing 64-bit applications, see *64-Bit Transition Guide*.

Command Line Primer

A command-line interface is a way for you to manipulate your computer in situations where a graphical approach is not available. The Terminal application is the Mac OS X gateway to the BSD command-line interface. Each window in Terminal contains a complete execution context, called a **shell**, that is separate from all other execution contexts. The shell itself is an interactive programming language interpreter, with a specialized syntax for executing commands and writing structured programs, called shell scripts.

Different shells feature slightly different capabilities and programming syntax. Although you can use any shell of your choice, the examples in this book assume that you are using the standard Mac OS X shell. The standard shell is `bash` if you are running Mac OS X v10.3 or later and `tcsh` if you are running an earlier version of the operating system.

The following sections provide some basic information and tips about using the command-line interface more effectively; they are not intended as an exhaustive reference for using the shell environments.

Basic Shell Concepts

Before you start working in any shell environment, there are some basic features of shell programming that you should understand. Some of these features are specific to Mac OS X, but many are common to all platforms that support shell programming.

Getting Information

At the command-line level, most documentation comes in the form of man pages. These are formatted pages that provide reference information for many shell commands, programs, and high-level concepts. To access one of these pages, you type the `man` command followed by the name of the thing you want to look up. For example, to look up information about the `bash` shell, you would type `man bash`. The man pages are also included in the ADC Reference Library. For more information, see *Mac OS X Man Pages*.

Note: Not all commands and programs have man pages. For a list of available man pages, look in the `/usr/share/man` directory.

Most shells have a command or man page that displays the list of built-in commands. Table A-1 lists the available shells in Mac OS X along with the ways you can access the list of built-in commands for the shell.

Table A-1 Getting a list of built-in commands

Shell	Command
`bash`	`help` or man bash
`sh`	`help` or man bash
`csh`	`builtins` or man csh
`tcsh`	`builtins` or man tcsh
`zsh`	man zshbuiltins

Specifying Files and Directories

Most commands in the shell operate on files and directories, the locations of which are identified by paths. The directory names that comprise a path are separated by forward-slash characters. For example, the path to the Terminal program is `/Applications/Utilities/Terminal.app`.

Table A-2 lists some of the standard shortcuts used to represent specific directories in the system. Because they are based on context, these shortcuts eliminate the need to type full paths in many situations.

Table A-2 Special path characters and their meaning

Path string	Description
.	A single period represents the current directory. This value is often used as a shortcut to eliminate the need to type in a full path. For example, the string "`./Test.c`" represents the `Test.c` file in the current directory.
..	Two periods represents the parent directory of the current directory. This string is used for navigating up one level from the current through the directory hierarchy. For example, the string "`../Test`" represents a sibling directory (named `Test`) of the current directory.
~	The tilde character represents the home directory of the currently logged-in user. In Mac OS X, this directory either resides in the local `/Users` directory or on a network server. For example, to specify the `Documents` directory of the current user, you would specify `~/Documents`.

File and directory names traditionally include only letters, numbers, a period (.), or the underscore character (_). Most other characters, including space characters, should be avoided. Although some Mac OS X file systems permit the use of these other characters, including spaces, you may have to add single or double quotation marks around any pathnames that contain them. For individual characters, you can also "escape" the character, that is, put a backslash character (\) immediately before the character in your string. For example, the path name `My Disk` would become either `"My Disk"` or `My\ Disk`.

Accessing Files on Volumes

On a typical UNIX system, the storage provided by local disk drives is coalesced into a single monolithic file system with a single root directory. This differs from the way the Finder presents local disk drives, which is as one or more volumes, with each volume acting as the root of its own directory hierarchy. To satisfy both worlds, Mac OS X includes a hidden directory `Volumes` at the root of the local file system. This directory contains all of the volumes attached to the local computer. To access the contents of other local volumes, you should always add the volume path at the beginning of the remaining directory information. For example, to access the `Applications` directory on a volume named `MacOSX`, you would use the path `/Volumes/MacOSX/Applications`

Note: To access files on the boot volume, you are not required to add volume information, since the root directory of the boot volume is `/`. Including the information still works, though, and is consistent with how you access other volumes. You must include the volume path information for all other volumes.

Flow Control

Many programs are capable of receiving text input from the user and printing text out to the console. They do so using the standard pipes (listed in Table A-3), which are created by the shell and passed to the program automatically.

Table A-3 Input and output sources for programs

Pipe	Description
`stdin`	The standard input pipe is the means through which data enters a program. By default, this is data typed in by the user from the command-line interface. You can also redirect the output from files or other commands to `stdin`.
`stdout`	The standard output pipe is where the program output is sent. By default, program output is sent back to the command line. You can also redirect the output from the program to other commands and programs.
`stderr`	The standard error pipe is where error messages are sent. By default, errors are displayed on the command line like standard output.

Redirecting Input and Output

From the command line you may redirect input and output from a program to a file or another program. You use the greater-than (>) character to redirect command output to a file and the less-than (<) character to use a file as input to the program. Redirecting file output lets you capture the results of running the command in the file system and store it for later use. Similarly, providing an input file lets you provide a program with preset input data, instead of requiring the user to type in that data.

In addition to file redirection, you can also redirect the output of one program to the input of another using the vertical bar (|) character. You can combine programs in this manner to implement more sophisticated versions of the same programs. For example, the command `man bash | grep "builtin commands"` redirects the formatted contents of the specified `man` page to the `grep` program, which searches those contents for any lines containing the word "commands". The result is a text listing of only those lines with the specified text, instead of the entire `man` page.

For more information about flow control, see the man page for the shell you are using.

Terminating Programs

To terminate the current running program from the command line, type Control-C. This keyboard shortcut sends an abort signal to the current command. In most cases this causes the command to terminate, although commands may install signal handlers to trap this command and respond differently.

Frequently Used Commands

Shell programming involves a mixture of built-in shell commands and standard programs that run in all shells. While most shells offer the same basic set of commands, there are often variations in the syntax and behavior of those commands. In addition to the shell commands, Mac OS X also provides a set of standard programs that run in all shells.

Table A-4 lists some of the more commonly used commands and programs. Because most of the items in this table are not built-in shell commands, you can use them from any shell. For syntax and usage information for each command, see the corresponding man page.

Table A-4 Frequently used commands and programs

Command	Meaning	Description
`cat`	Catenate	Catenates the specified list of files to `stdout`.
`cd`	Change Directory	A common shell command used to navigate the directory hierarchy.
`cp`	Copy	Copies files and directories (using the -r option) from one location to another.
`date`	Date	Displays the current date and time using the standard format. You can display this information in other formats by invoking the command with specific arguments.

Command	Meaning	Description
`echo`	Echo to Output	Writes its arguments to `stdout`. This command is most often used in shell scripts to print status information to the user.
`less`	Scroll Through Text	Used to scroll through the contents of a file or the results of another shell command. This command allows forward and backward navigation through the text.
`ls`	List	Displays the contents of the current directory. Specify the `-a` argument to list all directory contents (including hidden files and directories). Use the `-l` argument to display detailed information for each entry.
`mkdir`	Make Directory	Creates a new directory.
`more`	Scroll Through Text	Similar to the `less` command but more restrictive. Allows forward scrolling through the contents of a file or the results of another shell command.
`mv`	Move	Moves files and directories from one place to another. You also use this command to rename files and directories.
`pwd`	Print Working Directory	Displays the full path of the current directory.
`rm`	Remove	Deletes the specified file or files. You can use pattern matching characters (such as the asterisk) to match more than one file. You can also remove directories with this command, although use of `rmdir` is preferred.
`rmdir`	Remove Directory	Deletes a directory. The directory must be empty before you delete it.
Ctrl-C	Abort	Sends an abort signal to the current command. In most cases this causes the command to terminate, although commands may install signal handlers to trap this command and respond differently.

Environment Variables

Some programs require the use of environment variables for their execution. Environment variables are variables inherited by all programs executed in the shell's context. The shell itself uses environment variables to store information such as the name of the current user, the name of the host computer, and the paths to any executable programs. You can also create environment variables and use them to control the behavior of your program without modifying the program itself. For example, you might use an environment variable to tell your program to print debug information to the console.

To set the value of an environment variable, you use the appropriate shell command to associate a variable name with a value. For example, in the `tcsh` shell, to set the variable `MYFUNCTION` to the value `MyGetData`, you would type the following command in a Terminal window:

```
% setenv MYFUNCTION MyGetData
```

When you launch an application from a shell, the application inherits much of its parent shell's environment, including any exported environment variables. This form of inheritance can be a useful way to configure the application dynamically. For example, your application can check for the presence (or value) of an environment variable and change its behavior accordingly. Different shells support different semantics for exporting environment variables, so see the man page for your preferred shell for further information.

Although child processes of a shell inherit the environment of that shell, shells are separate execution contexts and do not share environment information with one another. Thus, variables you set in one Terminal window are not set in other Terminal windows. Once you close a Terminal window, any variables you set in that window are gone. If you want the value of a variable to persist between sessions and in all Terminal windows, you must set it in a shell startup script.

Another way to set environment variables in Mac OS X is with a special property list in your home directory. At login, the system looks for the following file:

```
~/.MacOSX/environment.plist
```

If the file is present, the system registers the environment variables in the property-list file. For more information on configuring environment variables, see *Runtime Configuration Guidelines*.

Running Programs

To run a program in the shell, you must type the complete pathname of the program's executable file, followed by any arguments, and then press the Return key. If a program is located in one of the shell's known directories, you can omit any path information and just type the program name. The list of known directories is stored in the shell's `PATH` environment variable and includes the directories containing most of the command-line tools.

For example, to run the `ls` command in the current user's home directory, you could simply type it at the command line and press the Return key.

```
host:~ steve$ ls
```

If you wanted to run a tool in the current user's home directory, however, you would need to precede it with the directory specifier. For example, to run the `MyCommandLineProgram` tool, you would use something like the following:

```
host:~ steve$ ./MyCommandLineProgram
```

To launch an application package, you can either use the `open` command (`open MyApp.app`) or launch the application by typing the pathname of the executable file inside the package, usually something like `./MyApp.app/Contents/MacOS/MyApp`.

Mac OS X Frameworks

This appendix contains information about the frameworks in the `/System/Library/Frameworks` directory. These frameworks provide the technologies needed to write software for Mac OS X. Table B-1 describes the frameworks and includes the first version of Mac OS X in which each became available. For umbrella frameworks, there are cross-references to later sections that list the subframeworks. Where applicable, this table also lists any key prefixes used by the classes, methods, functions, types, or constants of the framework. You should avoid using any of the specified prefixes in your own symbol names.

Table B-1 System frameworks

Name	First available	Prefixes	Description
`Accelerate.framework`	10.3	`cblas`, `vDSP`, `vv`	Umbrella framework for vector-optimized operations. See "Accelerate Framework" (page 100).
`AddressBook.framework`	10.2	`AB`, `ABV`	Contains functions for creating and accessing a systemwide database of contact information.
`AGL.framework`	10.0	`AGL`, `GL`, `glm`, `GLM`, `glu`, `GLU`	Contains Carbon interfaces for OpenGL.
`AppKit.framework`	10.0	`NS`	Contains classes and methods for the Cocoa user-interface layer. In general, link to `Cocoa.framework` instead of this framework.
`AppKit-Scripting.framework`	10.0	N/A	Deprecated. Use `AppKit.framework` instead.
`AppleScriptKit.framework`	10.0	`ASK`	Contains interfaces for creating AppleScript plug-ins and provides support for applications built with AppleScript Studio.
`AppleShare-Client.framework`	10.0	`AFP`	Contains interfaces for creating and parsing AFP URLs and for working with shared volumes.

Name	First available	Prefixes	Description
AppleShareClient-Core.framework	10.0	AFP	Contains utilities for handling URLs in AppleShare clients.
AppleTalk.framework	10.0	N/A	Do not use.
Application-Services.framework	10.0	AE, AX, ATSU, CG, CT, LS, PM, QD, UT	Umbrella framework for several application-level services. See "Application Services Framework" (page 101).
AudioToolbox.framework	10.0	AU, AUMIDI	Contains interfaces for getting audio stream data, routing audio signals through audio units, converting between audio formats, and playing back music.
AudioUnit.framework	10.0	AU	Contains interfaces for defining audio plug-ins.
Automator.framework	10.4	AM	Umbrella framework for creating Automator plug-ins. See "Automator Framework" (page 101).
Carbon.framework	10.0	HI, HR, ICA, ICD, Ink, Nav, OSA, PM, SFS,SR	Umbrella framework for Carbon-level services. See "Carbon Framework" (page 102).
Cocoa.framework	10.0	NS	Wrapper for including the Cocoa frameworks AppKit.framework, Foundation.framework, and CoreData.framework.
CoreAudio.framework	10.0	Audio	Contains the hardware abstraction layer interface for manipulating audio.
CoreAudioKit.framework	10.4	AU	Contains Objective-C interfaces for audio unit custom views.
CoreData.framework	10.4	NS	Contains interfaces for managing your application's data model.
CoreFoundation.framework	10.0	CF	Provides fundamental software services, including abstractions for common data types, string utilities, collection utilities, plug-in support, resource management, preferences, and XML parsing.
CoreMIDI.framework	10.0	MIDI	Contains utilities for implementing MIDI client programs.
CoreMIDIServer.framework	10.0	MIDI	Contains interfaces for creating MIDI drivers to be used by the system.

Name	First available	Prefixes	Description
CoreServices.framework	10.0	CF, MD, SK, WS	Umbrella framework for system-level services. See "Core Services Framework" (page 102).
CPlusTest.framework	10.4	None	Unit-testing framework for C++ code. (Requires the developer tools.)
Directory-Service.framework	10.0	ds	Contains interfaces for supporting network-based lookup and directory services in your application. You can also use this framework to develop directory service plug-ins.
DiscRecording.framework	10.2	DR	Contains interfaces for burning data to CDs and DVDs.
DiscRecording-UI.framework	10.2	DR	Contains the user interface layer for interacting with users during the burning of CDs and DVDs.
Disk-Arbitration.framework	10.4	DA	Contains interfaces for monitoring and responding to hard disk events.
DrawSprocket.framework	10.0	DSp	Contains the game sprocket component for drawing content to the screen.
DVComponent-Glue.framework	10.0	IDH	Contains interfaces for communicating with digital video devices, such as video cameras.
DVDPlayback.framework	10.3	DVD	Contains interfaces for embedding DVD playback features into your application.
Exception-Handling.framework	10.0	NS	Contains exception-handling classes for Cocoa applications.
ForceFeedback.framework	10.2	FF	Contains interfaces for communicating with force feedback–enabled devices.
Foundation.framework	10.0	NS	Contains the classes and methods for the Cocoa Foundation layer. If you are creating a Cocoa application, linking to the Cocoa framework is preferable.
FWAUserLib.framework	10.2	FWA	Contains interfaces for communicating with FireWire-based audio devices.
GLUT.framework	10.0	glut, GLUT	Contains interfaces for the OpenGL Utility Toolkit, which provides a platform-independent interface for managing windows.

Name	First available	Prefixes	Description
ICADevices.framework	10.3	ICD	Contains interfaces for communicating with digital devices such as scanners and cameras.
Installer-Plugins.framework	10.4	IFX	Contains interfaces for creating plug-ins that run during software installation sessions.
InstantMessage.framework	10.4	FZ, IM	Contains interfaces for obtaining the online status of an instant messaging user.
Interface-Builder.framework	10.0	IB	Contains interfaces for writing Interface Builder palettes.
IOBluetooth.framework	10.2	IO	Contains interfaces for communicating with Bluetooth devices.
IOBluetoothUI.framework	10.2	IO	Contains the user interface layer for interacting with users manipulating Bluetooth devices.
IOKit.framework	10.0	IO, IOBSD, IOCF	Contains the main interfaces for developing device drivers.
JavaEmbedding.framework	10.0	N/A	Do not use.
JavaVM.framework	10.0	JAWT, JDWP, JMM, JNI, JVMDI, JVMPI, JVMTI	Contains the system's Java Development Kit resources.
Kerberos.framework	10.0	GSS, KL, KRB, KRB5	Contains interfaces for using the Kerberos network authentication protocol.
Kernel.framework	10.0	*numerous*	Contains the BSD-level interfaces.
LDAP.framework	10.0	N/A	Do not use.
Message.framework	10.0	AS, MF, PO, POP, RSS, TOC, UR, URL	Contains Cocoa extensions for mail delivery.
OpenAL.framework	10.4	AL	Contains the interfaces for OpenAL, a cross-platform 3D audio delivery library.
OpenGL.framework	10.0	CGL, GL, glu, GLU	Contains the interfaces for OpenGL, which is a cross-platform 2D and 3D graphics rendering library.
OSAKit.framework	10.4	OSA	Contains Objective-C interfaces for managing and executing OSA-compliant scripts from your Cocoa applications.

Name	First available	Prefixes	Description
PCSC.framework	10.0	MSC, Scard, SCARD	Contains interfaces for interacting with smart card devices.
Preference-Panes.framework	10.0	NS	Contains interfaces for implementing custom modules for the System Preferences application.
Python.framework	10.3	Py	Contains the open source Python scripting language interfaces.
QTKit.framework	10.4	QT	Contains Objective-C interfaces for manipulating QuickTime content.
Quartz.framework	10.4	GF, PDF, QC, QCP	Umbrella framework for Quartz services. See "Quartz Framework" (page 103)
QuartzCore.framework	10.4	CI, CV	Contains the interfaces for Core Image and Core Video.
QuickTime.framework	10.0	N/A	Contains interfaces for embedding QuickTime multimedia into your application.
ScreenSaver.framework	10.0	N/A	Contains interfaces for writing screen savers.
Scripting.framework	10.0	NS	Deprecated. Use Foundation.framework instead.
Security.framework	10.0	CSSM, Sec	Contains interfaces for system-level user authentication and authorization.
Security-Foundation.framework	10.3	Sec	Contains Cocoa interfaces for authorizing users.
Security-Interface.framework	10.3	PSA, SF	Contains the user interface layer for authorizing users in Cocoa applications.
SenTestingKit.framework	10.4	Sen	Contains the interfaces for implementing unit tests in Objective-C. (Requires the developer tools.)
SyncServices.framework	10.4	ISync	Contains the interfaces for synchronizing application data with a central database.
System.framework	10.0	N/A	Do not use.
System-Configuration.framework	10.0	SC	Contains interfaces for accessing system-level configuration information.
Tcl.framework	10.3	Tcl	Contains interfaces for accessing the system's Tcl interpreter from an application.

Name	First available	Prefixes	Description
Tk.framework	10.4	Tk	Contains interfaces for accessing the system's Tk toolbox from an application.
TWAIN.framework	10.2	TW	Contains interfaces for accessing TWAIN-compliant image-scanning hardware.
vecLib.framework	10.0	N/A	Deprecated. Use Accelerate.framework instead. See "Accelerate Framework" (page 100).
WebKit.framework	10.2	DOM, Web	Umbrella framework for rendering HTML content. See "Web Kit Framework" (page 103).
Xgrid-Foundation.framework	10.4	XG	Contains interfaces for connecting to and managing computing cluster software.
XgridInterface.framework	10.4	N/A	Do not use.

Umbrella Framework Contents

Mac OS X contains several umbrella frameworks for major areas of functionality. Umbrella frameworks group several related frameworks into a larger framework that can be included in your project. When writing software, you must link against the umbrella framework itself; you cannot link against any of its subframeworks. The following sections describe the contents of the umbrella frameworks in Mac OS X.

Accelerate Framework

Table B-2 lists the subframeworks of the Accelerate framework (Accelerate.framework). This framework was introduced in Mac OS X version 10.3. If you are developing applications for earlier versions of Mac OS X, vecLib.framework is available as a standalone framework.

Table B-2 Subframeworks of the Accelerate framework

Subframework	Description
vecLib.framework	Contains vector-optimized interfaces for performing math, big-number, and DSP calculations, among others.
vImage.framework	Contains vector-optimized interfaces for manipulating image data.

Application Services Framework

Table B-3 lists the subframeworks of the Application Services framework (`ApplicationServices.framework`). These frameworks provide C-based interfaces and are intended primarily for Carbon applications, although other programs can use them. The listed frameworks are available in all versions of Mac OS X unless otherwise noted.

Table B-3 Subframeworks of the Application Services framework

Subframework	Description
`AE.framework`	Contains interfaces for creating and manipulating Apple events and making applications scriptable.
`ATS.framework`	Contains interfaces for font layout and management using Apple Type Services.
`ColorSync.framework`	Contains interfaces for color matching using ColorSync.
`CoreGraphics.framework`	Contains the Quartz interfaces for creating graphic content and rendering that content to the screen.
`FindByContent.framework`	Contains interfaces for searching the contents of files. Available in Mac OS X 10.1 and later.
`HIServices.framework`	Contains interfaces for accessibility, Internet Config, the pasteboard, the Process Manager, and the Translation Manager. Available in Mac OS X 10.2 and later.
`LangAnalysis.framework`	Contains the Language Analysis Manager interfaces.
`LaunchServices.framework`	Contains interfaces for launching applications.
`PrintCore.framework`	Contains the Carbon Printing Manager interfaces.
`QD.framework`	Contains the QuickDraw interfaces.
`SpeechSynthesis.framework`	Contains the Speech Manager interfaces.

Automator Framework

Table B-4 lists the subframeworks of the Automator framework (`Automator.framework`). This framework was introduced in Mac OS X version 10.4.

Table B-4 Subframeworks of the Automator framework

Subframework	Description
`MediaBrowser.framework`	Contains private interfaces for managing Automator plug-ins.

Carbon Framework

Table B-5 lists the subframeworks of the Carbon framework (`Carbon.framework`). The listed frameworks are available in all versions of Mac OS X unless otherwise noted.

Table B-5 Subframeworks of the Carbon framework

Subframework	Description
`CarbonSound.framework`	Contains the Sound Manager interfaces.
`CommonPanels.framework`	Contains interfaces for displaying the Font window, Color window, and some network-related dialogs.
`Help.framework`	Contains interfaces for launching and searching Apple Help.
`HIToolbox.framework`	Contains interfaces for the Carbon Event Manager, HIToolbox object, and other user interface–related managers.
`HTMLRendering.framework`	Contains interfaces for rendering HTML content. For Mac OS X version 10.2 and later, the Web Kit framework is the preferred framework for HTML rendering. See "Web Kit Framework" (page 103).
`ImageCapture.framework`	Contains interfaces for capturing images from digital cameras.
`Ink.framework`	Contains interfaces for managing pen-based input. (Ink events are defined with the Carbon Event Manager.) Available in Mac OS X version 10.3 and later.
`Navigation-Services.framework`	Contains interfaces for displaying file navigation dialogs.
`OpenScripting.framework`	Contains interfaces for writing scripting components and interacting with those components to manipulate and execute scripts.
`Print.framework`	Contains interfaces for displaying printing dialogs and extensions.
`SecurityHI.framework`	Contains interfaces for displaying security-related dialogs.
`Speech-Recognition.framework`	Contains the Speech Recognition Manager interfaces.

Core Services Framework

Table B-6 lists the subframeworks of the Core Services framework (`CoreServices.framework`). These frameworks provide C-based interfaces and are intended primarily for Carbon applications, although other programs can use them. The listed frameworks are available in all versions of Mac OS X unless otherwise noted.

Table B-6 Subframeworks of the Core Services framework

Subframework	Description
`CarbonCore.framework`	Contains interfaces for many legacy Carbon Managers.
`CFNetwork.framework`	Contains interfaces for network communication using HTTP, sockets, and Bonjour.
`NSLCore.framework`	Contains interfaces for the NSL Manager. Deprecated in Mac OS X version 10.2 and later.
`OSServices.framework`	Contains interfaces for Open Transport and many hardware-related legacy Carbon managers.
`OT.framework`	Contains interfaces for Open Transport. Deprecated in Mac OS X version 10.2 and later.
`SearchKit.framework`	Contains interfaces for the Search Kit. Available in Mac OS X version 10.3 and later.
`SecurityCore.framework`	Contains security-related interfaces. Deprecated in Mac OS X version 10.2 and later.
`WebServicesCore.framework`	Contains interfaces for accessing web-based services. Available in Mac OS X version 10.2 and later.

Quartz Framework

Table B-7 lists the subframeworks of the Quartz framework (`Quartz.framework`). This framework was introduced in Mac OS X version 10.4.

Table B-7 Subframeworks of the Quartz framework

Subframework	Description
`PDFKit.framework`	Contains Objective-C interfaces for displaying and managing PDF content in windows.
`QuartzComposer.framework`	Contains Objective-C interfaces for playing Quartz Composer compositions in an application.

Web Kit Framework

Table B-8 lists the subframeworks of the Web Kit framework (`WebKit.framework`). This framework was introduced in Mac OS X version 10.2.

Table B-8 Subframeworks of the Web Kit framework

Subframework	Description
`JavaScriptCore.framework`	Contains the library and resources for executing JavaScript code within an HTML page.
`WebCore.framework`	Contains the library and resources for rendering HTML content in an HTMLView control.

System Libraries

Note that some specialty libraries at the BSD level are not packaged as frameworks. Instead, Mac OS X includes many dynamic libraries in the `/usr/lib` directory and its subdirectories. Dynamic shared libraries are identified by their `.dylib` extension. Header files for the libraries are located in `/usr/include`.

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

Mac OS X Developer Tools

Apple provides a number of applications and command-line tools to help you develop your software. Many of these tools are installed with Mac OS X by default but some require you to install the Xcode Tools first. The Xcode Tools are available for free from the Apple Developer Connection website. For more information on how to get these tools, see "Getting the Xcode Tools" (page 12)

All of the applications described in the following sections can be found in the `/Developer/Applications` directory of your system. The command-line tools are located in the `/usr/bin` directory.

Note: Documentation for most of the command-line tools is available in the form of `man` pages. You can access these pages from the command line or from *Mac OS X Man Pages*. For more information about using command-line tools, see "Command Line Primer" (page 89).

Source Code Tools

Apple provides several applications and command-line tools for creating source code files.

Xcode

The centerpiece of the Xcode Tools is the Xcode application—an integrated developer environment (IDE) with the following features:

- A project management system for defining software products
- A code editing environment that includes features such as syntax coloring, code completion, and symbol indexing
- Visual design tools for creating your application's data model (see "Core Data" (page 66))
- An advanced documentation viewer for viewing and searching Apple documentation
- GCC compilers supporting C, C++, Objective-C, Objective-C++, and other compilers supporting Java and other languages
- Source-level debugging implemented using GDB and supporting features such as fix and continue and custom data formatters

- Distributed computing, enabling you to distribute large projects over several networked machines
- Predictive compilation that speeds single-file compile turnaround times
- Support for launching performance tools to analyze your software
- Support for integrated source-code management
- AppleScript support for automating the build process

For more information about Xcode features, see the *Xcode 2 User Guide* and the Xcode Release Notes.

Compilers, Linkers, Build Tools

Table C-1 lists the command-line compilers, linkers, and build tools. These tools are spread across several directories, including `/usr/bin`, `/Developer/Tools` and `/Developer/Private`.

Table C-1 Compilers, linkers, and build tools

Tool	Description
`as`	The Mac OS X Mach-O assembler.
`bsdmake`	The BSD make program.
`gcc`	The command-line interface to the GNU C compiler (GCC). Normally you invoke GCC through the Xcode application; however, you can execute it from a command line if you prefer.
`gnumake`	The GNU make program.
`jam`	An open-source build system initially released by Perforce, which provides the back-end for the Xcode application's build system. It is rarely used directly from the command line. Documented on the Perforce website at http://www.perforce.com/jam/jam.html.
`ld`	Combines several Mach-O (Mach object) files into one by combining like sections in like segments from all the object files, resolving external references, and searching libraries. Mach-O is the native executable format in Mac OS X.
`make`	A symbolic link to `gnumake`, the GNU make program. Note that the Xcode application automatically creates and executes make files for you; however the command-line make tools are available if you wish to use them.
`mkdep`	Constructs a set of include file dependencies. You can use this command in a make file if you are constructing make files instead of using Xcode to build and compile your program.
`pbprojectdump`	Takes an Xcode project (`.pbproj`) file and outputs a more nested version of the project structure. Note that, due to how conflicts are reflected in the project file, `pbprojectdump` cannot work with project files that have CVS conflicts.

Tool	Description
xcodebuild	Builds a target contained in an Xcode project. This command is useful if you need to build a project on another computer that you can connect to with Telnet. The xcodebuild tool reads your project file and builds it just as if you had used the Build command from within the Xcode application.

Library Utilities

Table C-2 lists the command-line tools available for creating libraries. These tools are all located in /usr/bin.

Table C-2 Tools for creating and updating libraries

Tool	Description
libtool	Takes object files and creates dynamically linked libraries or archive (statically linked) libraries, according to the options selected. The libtool command calls the ld command.
lorder	Determines interdependencies in a list of object files. The output is normally used to determine the optimum ordering of the object modules when a library is created so that all references can be resolved in a single pass of the loader.
ranlib	Adds to or updates the table of contents of an archive library.
redo_prebinding	Updates the prebinding of an executable or dynamic library when one of the dependent dynamic libraries changes.
update_prebinding	Updates prebinding information for libraries and executables when new files are added to the system.

Code Utilities

Table C-3 lists applications and command-line tools for manipulating source code and application resources. These tools are spread across several directories, including /usr/bin, Developer/Applications/Utilities, Developer/Applications/Graphics Toolsand /Developer/Tools.

Table C-3 Code utilities

Tool	Description
fpr	Reformats a Fortran file for printing by the UNIX line printer.
fsplit	Takes a Fortran multiple-routine source-code file and splits it into multiple files, one for each routine.
icns Browser	An application that displays the contents of .icns files. These files contain icon images at various sizes for use by the Finder and other applications.

Tool	Description
Icon Composer	An application that creates `.icns` files appropriate for use as application and document icons. You can import files, or drag images to specify the various data and mask components of the icon.
`ifnames`	Scans C source files and writes out a sorted list of all the identifiers that appear in `#if`, `#elif`, `#ifdef`, and `#ifndef` directives.
`indent`	Formats C source code.
`nmedit`	Changes global symbols in object code to static symbols. You can provide an input file that specifies which global symbols should remain global. The resulting object can still be used with the debugger.
OpenGL Shader Builder	An application that provides real-time entry, syntax checking, debugging, and analysis of vertex/fragment programs. It allows exporting of your creation to a sample GLUT application, which performs all necessary OpenGL setup, giving you a foundation to continue your application development. OpenGL is an open, cross-platform, three-dimensional (3D) graphics standard that supports the abstraction of current and future hardware accelerators. For more information about OpenGL, see *OpenGL Programming Guide for Mac OS X* in the Graphics & Imaging Documentation area.
`plutil`	Can check the syntax of a property list or convert it from one format to another (XML or binary).
`printf`	Formats and prints character strings and C constants.
Property List Editor	An application that lets you read and edit the contents of a property list. A property list, or plist, is a data representation used by Cocoa and Core Foundation as a convenient way to store, organize, and access standard object types. Property lists are useful when you need to store small amounts of persistent data. If you do use property lists, the `.plist` files must be bundled with other application files when you create your installation package.
`ResMerger`	Merges resources into resource files. When the Xcode application compiles Resource Manager resources, it sends them to a collector. After all Resource Manager resources have been compiled, the Xcode application calls `ResMerger` to put the resources in their final location.
`RezWack`	Takes a compiled resource (`.qtr`) file and inserts it together with the data fork (`.qtx` or `.exe` file) into a Windows application (`.exe`) file. The resulting file is a Windows application that has the sort of resource fork that QuickTime understands. You can use the `Rez` tool to compile a resource source (`.r`) file. The `RezWack` tool is part of the QuickTime 3 Software Development Kit for Windows.
`tops`	Performs universal search and replace operations on text strings in source files.
`unifdef`	Removes `#ifdef`, `#ifndef`, `#else`, and `#endif` lines from code as specified in the input options.
`UnRezWack`	Reverses the effects of `RezWack`; that is, converts a single Windows executable file into separate data and resource files.

Visual Design Tools

Apple provides several applications for creating programs, resources, and workflows visually. these tools are located in the various subdirectories of `/Developer/Applications`.

AppleScript Studio

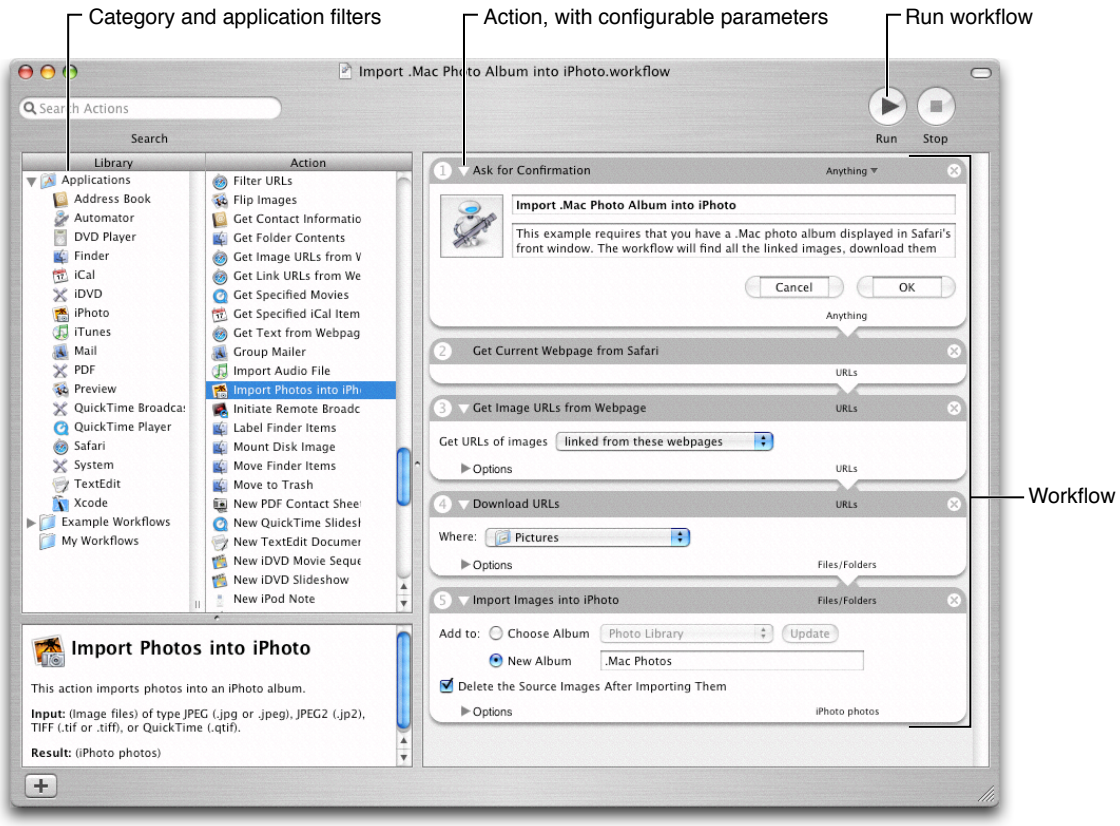
You can use AppleScript Studio to create AppleScript applications with complex user interfaces that support the Aqua human interface guidelines. AppleScript Studio is a combination of technologies, including AppleScript, Cocoa, the Xcode application, and Interface Builder.

For more information about AppleScript Studio, see *AppleScript Studio Programming Guide* or go to http://www.apple.com/applescript/studio/.

Automator

Introduced in Mac OS X version 10.4, Automator lets you automate common workflows on your computer without writing any code. The workflows you create can take advantage of many features of Mac OS X and any standard applications for which predefined **actions** are available. Actions are building blocks that represent tangible tasks, such as opening a file, saving a file, applying a filter, and so on. The output from one action becomes the input to another and you assemble the actions graphically with the Automator application. Figure C-1 shows the Automator main window and a workflow containing several actions.

Figure C-1 Automator main window



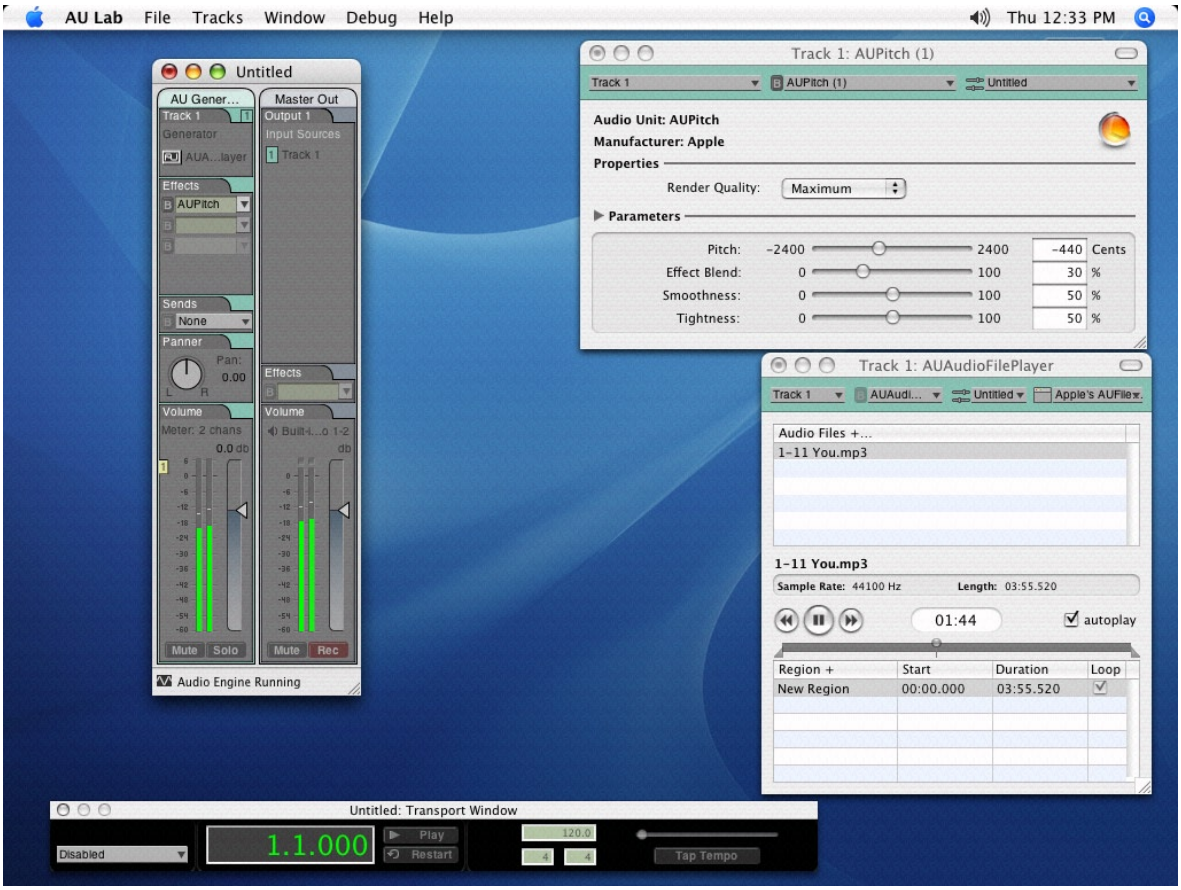
In cases where actions are not available for the tasks you want, you can often create them yourself. Automator supports the creation of actions using Objective-C code or AppleScript commands.

For more information about using Automator, see the Automator Help.

AU Lab

Introduced in Mac OS X version 10.4, AU Lab (Audio Unit Lab) lets you graphically host audio units and examine the results. You can use AU Lab to test the audio units you develop, do live mixing, and playback audio content. Parameters for the audio units are controlled graphically using the audio unit's custom interface or using a generic interface derived from the audio unit definition. Figure C-2 shows the AU Lab interface and some of the palettes for adjusting the audio parameters.

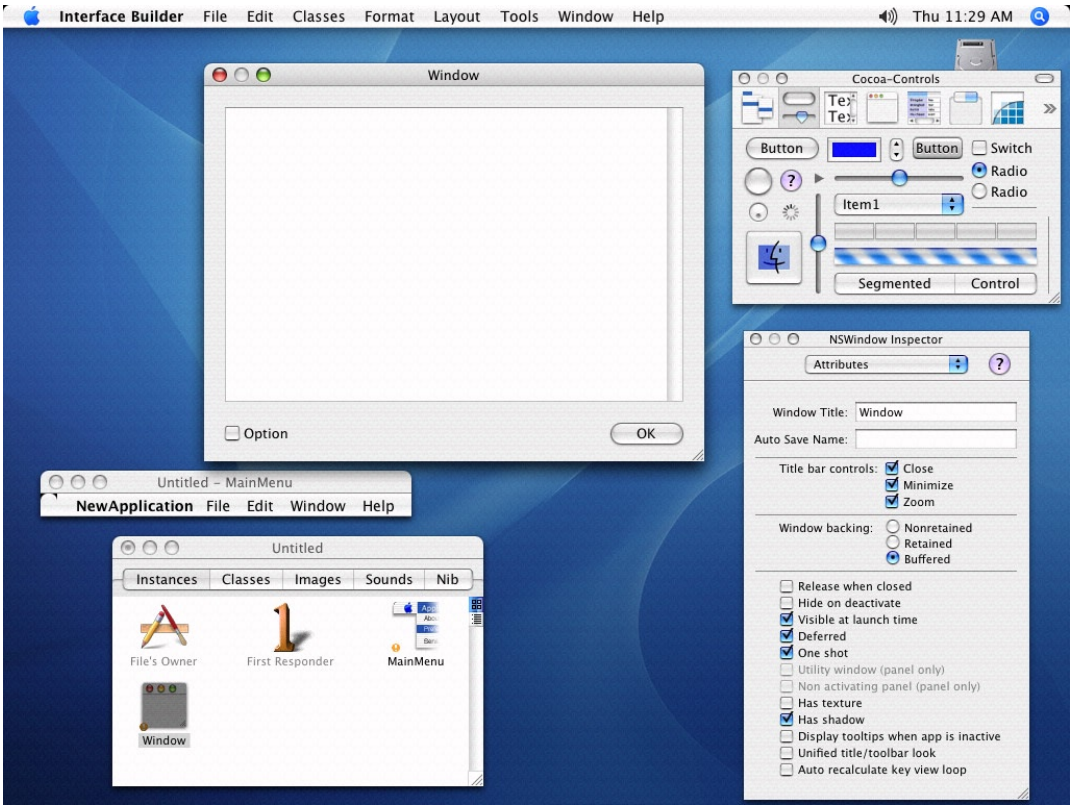
Figure C-2 AU Lab mixer and palettes



Interface Builder

Interface Builder is a graphical tool for designing the windows and menus of your applications. The tool provides guides to help with the layout and alignment of controls and other visual elements. You can create resource files (called **nib files**) for both Carbon and Cocoa applications and can localize the nib files for different languages.

Figure C-3 Interface Builder windows

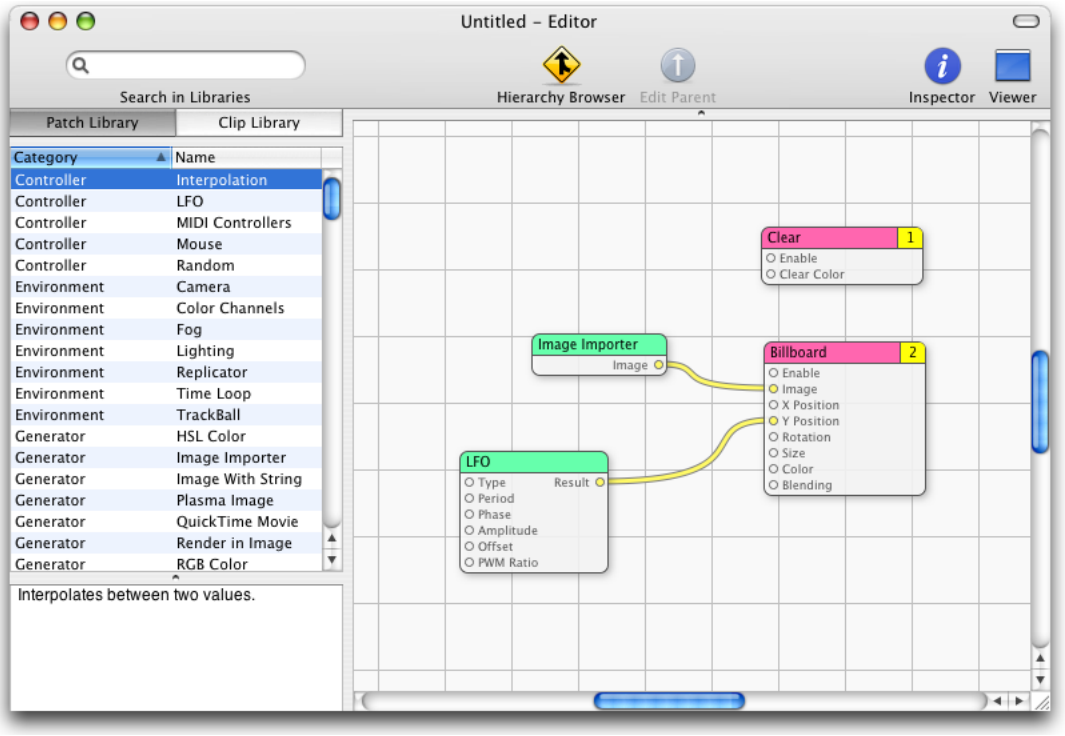


For more information on Interface Builder, see *Interface Builder*.

Quartz Composer

Introduced in Mac OS X version 10.4, Quartz Composer is a development tool for processing and rendering graphical data. Quartz Composer provides a visual development environment (Figure C-4) built on technologies such as Quartz 2D, Core Image, OpenGL, and QuickTime. You can use Quartz Composer as an exploratory tool to learn the tasks common to each visual technology without having to learn its application programming interface (API). In addition to supporting visual technologies, Quartz Composer also supports nongraphical technologies such as MIDI System Services and Rich Site Summary (RSS) file content.

Figure C-4 Quartz Composer editor window



For information on how to use Quartz composer, see *Quartz Composer Programming Guide*.

Debugging and Tuning Tools

Apple provides several tools for analyzing and monitoring the performance of your software. Performance should always be a key design goal of your programs. Using the provided tools, you can gather performance metrics and identify actual performance problems. You can then use this information to fix the problems and keep your software running efficiently.

General

Table C-4 lists the command-line tools available for debugging. These tools are all located in `/usr/bin`.

Table C-4 General debugging tools

Tool	Description
`defaults`	Lets you read, write, and delete Mac OS X user defaults. A Mac OS X application uses the defaults system to record user preferences and other information that must be maintained when the application is not running. Not all these defaults are necessarily accessible through the application's preferences.

Tool	Description
gdb	The GNU debugger. You can use it through the Xcode application or can invoke it directly from the command line.

Memory

Table C-5 lists the applications and command-line tools for debugging and tuning memory problems. These tools are spread across several directories, including /usr/bin and /Developer/Applications/Performance Tools.

Table C-5 Memory debugging and tuning tools

Tool	Description
heap	Lists all the objects currently allocated on the heap of the current process. It also describes any Objective-C objects, listed by class.
leaks	Examines a specified process for malloc-allocated buffers that are not referenced by the program.
MallocDebug	A utility application for understanding how an application uses memory. You can use MallocDebug to measure and analyze all allocated memory in an application or to measure the memory allocated since a given time. MallocDebug also detects memory leaks.
malloc_history	Inspects a given process and lists the malloc allocations performed by it. This tool relies on information provided by the standard malloc library when debugging options have been turned on. If you specify an address, malloc_history lists the allocations and deallocations that have manipulated a buffer at that address. For each allocation, a stack trace describing who called malloc or free is listed.
ObjectAlloc	An application that allows you to observe memory allocation and deallocation activity in an application, almost in real time. It also keeps a history of memory allocations, so you can scroll backward and forward in time. This process can help identify some types of allocation patterns.
vmmap	Displays the virtual memory regions allocated in a specified process, helping you understand how memory is being used and the purpose of memory (text segment, data segment, and so on) at a given address.
vm_stat	Displays Mach virtual memory statistics.

Graphics

Table C-6 lists the applications for debugging graphics problems. These tools are in either /Developer/Applications/Performance Tools or /Developer/Applications/Graphics Tools.

Table C-6 Graphics tools

Tool	Description
OpenGL Driver Monitor	An application that displays extensive information about the OpenGL environment.
OpenGL Profiler	An application that creates a runtime profile of an OpenGL-based application. The profile contains OpenGL function-call timing information, a listing of all the OpenGL function calls your application made, and all the OpenGL-related data needed to replay your profiling session.
Pixie	A magnifying glass utility for Mac OS X. Pixie is useful for doing pixel-perfect layout, checking the correctness of graphics and user interface elements, and getting magnified screen shots.
Quartz Debug	A debugging utility for the Quartz graphics system. Quartz Debug is documented in *Drawing Performance Guidelines*.

Examining Code

Table C-7 lists the applications and command-line tools for examining generated code files. These tools are all located in `/usr/bin`.

Table C-7 Tools for examining code

Tool	Description
`c2ph`	Parses C code and outputs debugger information in the Stabs format, showing offsets of all the members of structures. For information on Stabs, see `/Developer/ADC Reference Library/documentation/DeveloperTools/gdb/stabs/stabs_toc.html`
`cscope`	An interactive command-line tool that allows the user to browse through C source files for specified elements of code, such as functions, function calls, macros, variables, and preprocessor symbols.
`ctags`	Makes a tags file for the `ex` line editor from specified C, Pascal, Fortran, YACC, Lex, or Lisp source files. A tags file lists the locations of symbols such as subroutines, typedefs, structs, enums, unions, and #defines.
`error`	Analyzes error messages and can open a text editor to display the source of the error. The `error` tool is run with its input connected via a pipe to the output of the compiler or language processor generating the error messages. Note that the service provided by the `error` command is built into the Xcode application.
`nibtool`	Lets you print, update, and verify the contents of a nib file from the command line.
`nm`	Displays the symbol tables of one or more object files, including the symbol type and value for each symbol.
`otool`	Displays specified parts of object files or libraries.

Tool	Description
`pagestuff`	Displays information about the specified logical pages of a file conforming to the Mach-O executable format. For each specified page of code, `pagestuff` displays symbols (function and static data structure names).
`pstruct`	An alias to `c2ph`.
`strings`	Looks for ASCII strings in an object file or other binary file.

Performance

Table C-8 lists the applications and command-line tools for analyzing and monitoring performance. For information about performance and the available performance tools, see *Performance Overview*. These tools are spread across several directories, including `/usr/bin` and `/Developer/Applications/Performance Tools`.

Table C-8 Performance tools

Tool	Description
`gprof`	Produces an execution profile of a C, Pascal, or Fortran77 program. The tool lists the total execution times and call counts for each of the functions in the application, and sorts the functions according to the time they represent including the time of their call graph descendents.
MONster	An application that collects data from processor and memory controller performance counters and presents it in tabular form. It can collect data systemwide, or from a specific program running on the system.
`sample`	Gathers data about the running behavior of a process. The `sample` tool stops the process at user-defined intervals, records the current function being executed by the process, and checks the stack to find how the current function was called. It then lets the application continue. At the end of a sampling session, sample produces a report showing which functions were executing during the session.
Sampler	A utility you can use to help understand an application's running behavior. Sampler can identify the routines in which the program spends most time executing. It can also summarize why allocation routines, system calls, or arbitrary functions were called.
Shark	An application that profiles the system to see how time is being spent. It can work at the system, task, or thread level and can correlate performance counter events with source code. Shark's histogram view can be used to observe scheduling and other time-dependent behavior. It can produce profiles of hardware and software performance events such as cache misses, virtual memory activity, instruction dependency stalls, and so forth.
Thread Viewer	A utility for graphically displaying activity across a range of threads. It provides timeline color-coded views of activity on each thread. By clicking a point on a timeline, you can see a sample backtrace of activity at that time.

Tool	Description
top	Displays an ongoing sample of system-use statistics. It can operate in various modes, but by default shows CPU and memory use for each process in the system.

KEXTs, Drivers, and Instruction Traces

Table C-8 lists the applications and command-line tools for working with hardware and kernel-level programs. These tools are spread across several directories, including /usr/bin, /Developer/Applications/Utilities and /Developer/Applications/Performance Tools.

Table C-9 KEXT, driver, and instruction trace tools

Tool	Description
USB Prober	An application that displays detailed information about all the USB ports and devices on the system.
Reggie SE	An application that examines and modifies CPU and PCI configuration registers in PowerPC processors.
ktrace	Enables kernel trace logging for specified processes. The kernel operations that are traced include system calls, namei translations, signal processing, and I/O.
kdump	Displays the kernel trace files produced by ktrace in human readable format.
acid	Analyzes TT6E (but not TT6) instruction traces and presents detailed analyses and histogram reports.
amber	Captures the instruction and data address stream generated by a process running in Mac OS X and saves it to disk in TT6, TT6E, or FULL format. Custom trace filters can be built using the amber_extfilt.a module in /Developer/Examples/CHUD/Amber/ExternalTraceFilter/. Differences between TT6 and TT6E format as well as the specifics of the FULL trace format are detailed in *Amber Trace Format Specification v1.1* (/Developer/ADC Reference Library/CHUD/AmberTraceFormats.pdf).
simg4	A cycle-accurate simulator of the Motorola 7400 processor that takes TT6 (not TT6E) traces as input.
simg5	A cycle-accurate simulator of the IBM 970 processor that takes TT6 (not TT6E) traces as input.

Documentation and Help Tools

Table C-10 lists applications and command-line tools for creating or working with documentation and online help. Except where noted, these tools are all located in /usr/bin.

Table C-10 Documentation and help tools

Tool	Description
Apple Help Indexing Tool	An application to create a search index for a help file. The Apple Help Viewer uses the Sherlock search engine to provide fast, full-text searching of Apple Help files. Instructions for creating Apple Help and for using the indexing tool are in *Providing User Assistance With Apple Help* (`/Developer/ADC Reference Library/documentation/Carbon/Conceptual/ProvidingUserAssitAppleHelp/`).
`compileHelp`	Merges contextual help RTF snippets into one file. This tool is included to support legacy applications. New contextual help projects do not use this tool.
`gatherheaderdoc`	Gathers HeaderDoc output, creating a single index page and cross-links between documents.
`headerdoc2HTML`	Generates HTML documentation from structured commentary in C, C++, and Objective-C header files. The HeaderDoc tags and scripts are described at http://developer.apple.com/darwin/projects/headerdoc/.
`pbhelpindexer`	Indexes documentation created with HeaderDoc so that the Xcode application can find the appropriate excerpt for a given symbol. The HeaderDoc tags and scripts are described at http://developer.apple.com/darwin/projects/headerdoc/.
`install-info`	Inserts menu entries from an Info file into the top-level dir file in the GNU Texinfo documentation system. It's most often run as part of software installation or when constructing a dir file for all manuals on a system. See http://www.gnu.org/software/texinfo/manual/texinfo/ for more information on the GNU Texinfo system.

Localization Tools

Table C-11 lists the applications and command-line tools for localizing your own applications. Except for AppleGlot, these tools are spread across several directories, including `/usr/bin` and `Developer/Tools`.

Table C-11 Localization tools

Tool	Description
AppleGlot	A localization application for Mac OS X software. AppleGlot can extract localizable information from Mac OS X software put it into a text file for translation. AppleGlot then reinserts these strings in the application. AppleGlot also supports incremental localization; once software has been localized, subsequent use of AppleGlot extracts only the new or changed items in the file that potentially require localization. AppleGlot supports Cocoa and Carbon applications, bundled applications, and shared libraries or frameworks. It works with nib files, Resource Manager `.rsrc` files, and `.string` files in XML format or with an AppleGlot-defined key-value format. For information on where to find AppleGlot, go to http://developer.apple.com/intl/localization/tools.html.

Tool	Description
DeRez	Decompiles the resource fork of a resource file according to the type declarations in the type declaration files you specify. You can use this utility to find strings for localization purposes, for example. DeRez works with Resource Manager resource files, not with nib files.
`genstrings`	Takes the strings from C source code (`NSLocalizedString...`, `CFCopyLocalized-String...` functions) and generates string table files (`.strings` files). This tool can also work with `Bundle.localizedString...` methods in Java.
`Rez`	Compiles the resource fork of a file according to the textual description contained in the resource description files. You can use Rez to recompile the resource files you decompiled with DeRez after you have localized the strings.

Version Control Tools

Apple provides command-line tools to support several version-control systems. Unless otherwise noted, these tools are located in `/usr/bin`.

RCS

Table C-12 lists the command-line tools to use with the RCS system.

Table C-12 RCS tools

Tool	Description
`ci`	Stores revisions in RCS files. If the RCS file doesn't exist, `ci` creates one.
`co`	Retrieves a revision from an RCS file and stores it in the corresponding working file.
`rcs`	Creates new RCS files or changes attributes of existing ones.
`rcs-checkin`	Checks a file into a new RCS file and uses the file's first line for the description.
`rcs2log`	Generates a change log from RCS files—which can possibly be located in a CVS repository—and sends the change log to standard output.
`rcsclean`	Compares the working file to the latest revision (or a specified revision) in the corresponding RCS file and removes the working file if there is no difference.
`rcsdiff`	Compares two revisions of an RCS file or the working file and one revision.
`rcsmerge`	Merges the changes in two revisions of an RCS file into the corresponding working file.

CVS

Table C-13 lists the command-line tools to use with the Concurrent Versions System (CVS) source control system.

Table C-13 CVS tools

Tool	Description
`agvtool`	Speeds up common versioning operations for Xcode projects that use the Apple-generic versioning system. It automatically embeds version information in the products produced by the Xcode application and performs certain CVS operations such as submitting the project with a new version number. The `agvtool` program has no man page; for documentation, enter `agvtool help` in the Terminal.
`cvs`	The latest tool for managing information in the CVS repository. (Note, this tool does not support CVS wrappers.) See the `cvs` man page for details. See also, `ocvs` below.
`cvs-wrap`	Wraps a directory into a GZIP format tar file. This single file can be handled more easily by CVS than the original directory.
`cvs-unwrap`	Extracts directories from a GZIP format tar file created by `cvs-wrap`.
`ocvs`	An older version of the `cvs` tool that still supports CVS wrappers. See the `ocvs` man page for details.

Comparing Files

Table C-14 lists the command-line tools for comparing files.

Table C-14 Comparison tools

Tool	Description
`diff`	Compares two files or the files in two directories.
`diff3`	Compares three files.
`diffpp`	Annotates the output of `diff` so that it can be printed with GNU `enscript`. This enables `enscript` to highlight the modified portions of the file.
`diffstat`	Reads one or more files output by `diff` and displays a histogram of the insertions, deletions, and modifications per file.
FileMerge	An application that compares two ASCII files or two directories. For a more accurate comparison, you can compare two files or directories to a common ancestor. After comparing, you can merge the files or directories.
`merge`	Compares two files modified from the same original file and then combines all the changes into a single file. The `merge` tool warns you if both modified files have changes in the same lines.

Tool	Description
patch	Takes the output of diff and applies it to one or more copies of the original, unchanged file to create patched versions of the file.
sdiff	Compares two files and displays the differences so you can decide how to resolve them interactively. It then writes the results out to a file. A command-line version of FileMerge.

Packaging Tools

Table C-15 lists the applications and command-line tools used for packaging applications. These tools are spread across several directories, including /usr/bin, /Developer/Tools, and /Developer/Applications.

Table C-15 Packaging tools

Tool	Description
CpMac	Copies a file or a directory, including subdirectories, preserving metadata and forks.
GetFileInfo	Gets the file attributes of files in an HFS+ directory.
install	Copies files to a target file or directory. Unlike the cp or mv commands, the install command lets you specify the new copy's owner, group ID, file flags, and mode.
Installer	The native installer for Mac OS X.
install_name_tool	Changes the dynamic shared library install names recorded in a Mach-O binary.
lipo	Can create a multiple-architecture ("fat") executable file from one or more input files, list the architectures in a fat file, create a single-architecture file from a fat file, or make a new fat file with a subset of the architectures in the original fat file.
MergePef	Merges two or more PEF files into a single file. PEF format is used for Mac OS 9 code.
mkbom	Creates a bill of materials for a directory.
MvMac	Moves files, preserving metadata and forks.
PackageMaker	An application that you can use to create an installation package. When the user double-clicks the package, the Installer launches and installs the files you packaged. You can use PackageMaker to package files or to assemble individual packages into a single package. You can run PackageMaker from the command line. For details, enter the following command on the command line: /Developer/Applications/PackageMaker.app/Contents/MacOS/PackageMaker. Be sure to use full paths in all command-line arguments.

Tool	Description
SetFile	Sets the attributes of files in an HFS+ directory.
SplitForks	Removes the resource fork in a file or all the resource forks in the files in a specified directory and saves them alongside the original files as hidden files (a hidden file has the same name as the original file, except that it has a "dot-underscore" prefix; for example ._MyPhoto.jpg.).

Scripting Tools

The tools listed in the following sections are spread across the directories /usr/bin and /Developer/Applications.

AppleScript Studio

You can use AppleScript Studio to create AppleScript applications with complex user interfaces that support the Aqua human interface guidelines. AppleScript Studio is a combination of technologies, including AppleScript, Cocoa, the Xcode application, and Interface Builder.

For more information about AppleScript Studio, see *AppleScript Studio Programming Guide* or go to http://www.apple.com/applescript/studio/.

Interpreters and Compilers

Table C-16 lists the command-line script interpreters and compilers.

Table C-16 Script interpreters and compilers

Tool	Description
awk	A pattern-directed scripting language for scanning and processing files. The scripting language is described on the awk man page.
osacompile	Compiles the specified files, or standard input, into a single script. Input files may be plain text or other compiled scripts. The osacompile command works with AppleScript and with any other OSA scripting language.
osascript	Executes a script file, which may be plain text or a compiled script. The osascript command works with AppleScript and with any other scripting language that conforms to the Open Scripting Architecture (OSA).
perl	Executes scripts written in the Practical Extraction and Report Language (Perl). The man page for this command introduces the language and gives a list of other man pages that fully document it.
perlcc	Compiles Perl scripts.

Tool	Description
python	The interpreter for the Python language, an interactive, object-oriented language. Use the pydoc command to read documentation on Python modules.
ruby	The interpreter for the Ruby language, an interpreted object-oriented scripting language.
sed	Reads a set of files and processes them according to a list of commands.
tclsh	A shell-like application that interprets Tcl commands. It runs interactively if called without arguments. Tcl is a scripting language, like Perl, Python, or Ruby. However, Tcl is usually embedded and thus called from the Tcl library rather than by an interpreter such as tclsh.

Script Language Converters

Table C-17 lists the available command-line script language converters.

Table C-17 Script language converters

Tool	Description
a2p	Converts an awk script to a Perl script.
s2p	Converts a sed script to a Perl script.

Perl Tools

Table C-18 lists the available command-line Perl tools.

Table C-18 Perl tools

Tool	Description
dprofpp	Displays profile data generated for a Perl script by a Perl profiler.
find2perl	Converts find command lines to equivalent Perl code.
h2ph	Converts C header files to Perl header file format.
h2xs	Builds a Perl extension from C header files. The extension includes functions that can be used to retrieve the value of any #define statement that was in the C header files.
perlbug	An interactive tool that helps you report bugs for the Perl language.
perldoc	Looks up and displays documentation for Perl library modules and other Perl scripts that include internal documentation. If a man page exists for the module, you can use man instead.

Tool	Description
pl2pm	Aids in the conversion of Perl 4 .pl library files to Perl 5 library modules. This tool is useful if you plan to update your library to use some of the features new in Perl 5.
splain	Forces verbose warning diagnostics by the Perl compiler and interpreter.

Parsers and Lexical Analyzers

Table C-19 lists the available command-line parsers and lexical analyzers.

Table C-19 Parsers and lexical analyzers

Tool	Description
bison	Generates parsers from grammar specification files. A somewhat more flexible replacement for yacc.
flex	Generates programs that scan text files and perform pattern matching. When one of these programs matches the pattern, it executes the C routine you provide for that pattern.
lex	An alias for flex.
yacc	Generates parsers from grammar specification files. Used in conjunction with flex to created lexical analyzer programs.

Documentation Tools

Table C-20 lists the available command-line scripting documentation tools.

Table C-20 Scripting documentation tools

Tool	Description
pod2html	Converts files from pod format to HTML format. The pod (Plain Old Documentation) format is defined in the perlpod man page.
pod2latex	Converts files from pod format to LaTeX format. LaTeX is a document preparation system built on the TeX text formatter.
pod2man	Converts files from pod format to *roff code, which can be displayed using nroff via man, or printed using troff.
pod2text	Converts pod data to formatted ASCII text.
pod2usage	Similar to pod2text, but can output just the synopsis information or the synopsis plus any options/arguments sections instead of the entire man page.
podchecker	Checks the syntax of documentation files that are in pod format and outputs errors to standard error.

Tool	Description
`podselect`	Prints selected sections of pod documentation to standard output.

Java Tools

The tools listed in the following sections are spread across the directories `/usr/bin` and `/Developer/Applications/Java Tools`.

General

Table C-21 lists the command-line tools used for building, debugging, and running Java programs.

Table C-21 Java tools

Tool	Description
`java`	Starts the Java runtime environment and launches a Java application.
`javac`	The standard Java compiler from Sun Microsystems.
`jdb`	The Java debugger. It provides inspection and debugging of a local or remote Java virtual machine.
`jikes`	A Java compiler from IBM, which is faster than `javac` for many applications.

Java Utilities

Table C-22 lists some of the applications and command-line tools for working with Java.

Table C-22 Java utilities

Tool	Description
`idlj`	Reads an Object Management Group (OMG) Interface Definition Language (IDL) file and translates it, or maps it, to a Java interface. The `idlj` compiler also creates stub, skeleton, helper, holder, and other files as necessary. These Java files are generated from the IDL file according to the mapping specified in the OMG document *OMG IDL to Java Language Mapping Specification, formal, 99-07-53*. The `idlj` compiler is documented at http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/toJavaPortableUG.html. IDL files are used to allow objects from different languages to interact with a common Object Request Broker (ORB), allowing remote invocation between languages.
JavaBrowser	An application that lets you browse Java classes, documentation, and source files.

Tool	Description
`javadoc`	Parses the declarations and documentation comments in a set of Java source files and produces HTML pages describing the public and protected classes, inner classes, interfaces, constructors, methods, and fields.
`javah`	Generates C header and source files from Java classes. The generated header and source files are used by C programs to reference instance variables of a Java object so that you can call Java code from inside your Mac OS X native application.
`native2ascii`	Converts characters that are not in Latin-1 or Unicode encoding to ASCII for use with `javac` and other Java tools. It also can do the reverse conversion of Latin-1 or Unicode to native-encoded characters.
`rmic`	A compiler that generates stub and skeleton class files for remote objects from the names of compiled Java classes that contain remote object implementations. A remote object is one that implements the interface `java.rmi.Remote`.
`rmiregistry`	Creates and starts a remote object registry. A remote object registry is a naming service that makes it possible for clients on the host to look up remote objects and invoke remote methods.

Java Archive (JAR) Files

Table C-23 lists the available JAR file applications and command-line tools.

Table C-23 JAR file tools

Tool	Description
`extcheck`	Checks a specified JAR file for title and version conflicts with any extensions installed in the Java Developer Kit software.
`jar`	Combines and compresses multiple files into a single Java archive (JAR) file so they can be downloaded by a Java agent (such as a browser) in a single HTTP transaction.
Jar Bundler	An application that allows you to package your Java program's files and resources into a single double-clickable application bundle. Jar Bundler lets you modify certain properties so your Java application behaves as a better Mac OS X citizen and lets you specify arguments sent to the Java virtual machine (VM) when the application starts up.
`jarsigner`	Lets you sign JAR files and verify the signatures and integrity of signed JAR files.

Kernel Extension Tools

Table C-24 lists the command-line tools that are useful for kernel extension development. These tools are all located in either `/sbin` or `/usr/sbin`.

Table C-24 Kernel extension tools

Tool	Description
`kextload`	Loads kernel extensions, validates them to make sure they can be loaded by other mechanisms, and generates symbol files for debugging them.
`kextstat`	Displays the status of any kernel extensions currently loaded in the kernel.
`kextunload`	Terminates and unregisters I/O Kit objects associated with a KEXT and unloads the code for the KEXT.

I/O Kit Driver Tools

Table C-25 lists the applications and command-line tools for developing device drivers. These tools are spread across the directories `/usr/sbin` and `/Developer/Applications/Utilities`.

Table C-25 Driver tools

Tool	Description
I/O Registry Explorer	An application that you can use to examine the configuration of devices on your computer. I/O Registry Explorer (filename `IORegistryExplorer`) provides a graphical representation of the I/O Registry tree. Documentation for I/O Registry Explorer is at *I/O Kit Fundamentals* in the Device Drivers & I/O Kit Documentation area.
`ioreg`	A command-line version of I/O Registry Explorer. The `ioreg` tool displays the tree in a Terminal window, allowing you to cut and paste sections of the tree.
`ioalloccount`	Displays a summary of memory allocated by I/O Kit allocators listed by type (instance, container, and `IOMalloc`). This tool is useful for tracking memory leaks.
`ioclasscount`	Shows the number of instances allocated for each specified class. This tool is also useful for tracking memory leaks.

Document Revision History

This table describes the changes to *Mac OS X Technology Overview*.

Date	Notes
2006-06-28	Associated in-use prefix information with the system frameworks. Clarified directories containing developer tools.
2005-10-04	Added references to "Universal Binary Programming Guidelines."
2005-08-11	Fixed minor typos. Updated environment variable inheritance information.
2005-07-07	Incorporated developer feedback.
	Added AppleScript to the list of application environments.
2005-06-04	Corrected the man page name for SQLite.
2005-04-29	Fixed broken links and incorporated user feedback.
	Incorporated porting and technology guidelines from "Apple Software Design Guidelines." Added information about new system technologies. Changed "Rendezvous" to "Bonjour."
	Added new software types to list of development opportunities.
	Added a command-line primer.
	Added a summary of the available development tools.
	Updated the list of system frameworks.
2004-05-27	First version of *Mac OS X Technology Overview*. Some of the information in this document previously appeared in *System Overview*.

Document Revision History

Glossary

abstract type Defines, in information property lists, general characteristics of a family of documents. Each abstract type has corresponding concrete types. See also "concrete type".

actions Building blocks used to build workflows in Automator.

active window The frontmost modal or document window. Only the contents of the active window are affected by user actions. The active window has distinctive details that aren't visible for inactive windows.

Address Book A technology for managing names, addresses, phone numbers, and other contact-related information. Mac OS X provides the Address Book application for managing contact data. It also provides the Address Book framework so that applications can programmatically manage the data.

address space Describes the range of memory (both physical and virtual) that a process uses while running. In Mac OS X, processes do not share address space.

alias A lightweight reference to files and folders in Mac OS Standard (HFS) and Mac OS Extended (HFS+) file systems. An alias allows multiple references to files and folders without requiring multiple copies of these items. Aliases are not as fragile as symbolic links because they identify the volume and location on disk of a referenced file or folder; the file or folder can be moved around without breaking the alias. See also "symbolic link".

anti-aliasing A technique that smooths the roughness in images or sound caused by aliasing. During frequency sampling, aliasing generates a false (alias) frequency along with the correct one. With images this produces a stair-step effect. Anti-aliasing corrects this by adjusting pixel positions or setting pixel intensities so that there is a more gradual transition between pixels.

Apple event A high-level operating-system event that conforms to the Apple Event Interprocess Messaging Protocol (AEIMP). An Apple event typically consists of a message from an application to itself or to another application.

AppleTalk A suite of network protocols that is standard on Macintosh computers and can be integrated with other network systems, such as the Internet.

Application Kit A Cocoa framework that implements an application's user interface. The Application Kit provides a basic program structure for applications that draw on the screen and respond to events.

application packaging Putting code and resources in the prescribed directory locations inside application bundles. "Application package" is sometimes used synonymously with "application bundle."

Aqua A set of guidelines that define the appearance and behavior of Mac OS X applications. The Aqua guidelines bring a unique look to applications, integrating color, depth, clarity, translucence, and motion to present a vibrant appearance. If you use Carbon, Cocoa, or X11 to create your application's interface, you get the Aqua appearance automatically

ASCII American Standard Code for Information Interchange. A 7-bit character set (commonly represented using 8 bits) that defines 128 unique character codes. See also "Unicode".

bit depth The number of bits used to describe something, such as the color of a pixel. Each additional bit in a binary number doubles the number of possibilities.

bitmap A data structure that represents the positions and states of a corresponding set of pixels.

Bonjour Apple's technology for zero-configuration networking. Bonjour enables dynamic discovery of services over a network.

BSD Berkeley Software Distribution. Formerly known as the Berkeley version of UNIX, BSD is now simply called the BSD operating system. BSD provides low-level features such as networking, thread management, and process communication. It also includes a command-shell environment for managing system resources. The BSD portion of Mac OS X is based on version 5 of the FreeBSD distribution.

buffered window A window with a memory buffer into which all drawing is rendered. All graphics are first drawn in the buffer, and then the buffer is flushed to the screen.

bundle A directory in the file system that stores executable code and the software resources related to that code. Applications, plug-ins, and frameworks are types of bundles. Except for frameworks, bundles are file packages, presented by the Finder as a single file.

bytecode Computer object code that is processed by a virtual machine. The virtual machine converts generalized machine instructions into specific machine instructions (instructions that a computer's processor can understand). Bytecode is the result of compiling source language statements written in any language that supports this approach. The best-known language today that uses the bytecode and virtual machine approach is Java. In Java, bytecode is contained in a binary file with a `.class` suffix. (Strictly speaking, "bytecode" means that the individual instructions are one byte long, as opposed to PowerPC code, for example, which is four bytes long.) See also "virtual machine (VM)".

Carbon An application environment in Mac OS X that features a set of procedural programming interfaces derived from earlier versions of the Mac OS. The Carbon API has been modified to work properly with Mac OS X, especially with the foundation of the operating system, the kernel environment. Carbon applications can run in Mac OS X and Mac OS 9.

CFM Code Fragment Manager, the library manager and code loader for processes based on PEF (Preferred Executable Format) object files (in Carbon).

class In object-oriented languages such as Java and Objective-C, a prototype for a particular kind of object. A class definition declares instance variables and defines methods for all members of the class. Objects that belong to the same class have the same types of instance variables and have access to the same methods (included the instance variables and methods inherited from superclasses).

Classic An application environment in Mac OS X that lets you run non-Carbon legacy Mac OS software. It supports programs built for both PowerPC and 68000-family chip architectures and is fully integrated with the Finder and the other application environments.

Clipboard A per-user server (also known as the pasteboard) that enables the transfer of data between applications, including the Finder. This server is shared by all running applications and contains data that the user has cut or copied, as well as other data that one application wants to transfer to another, such as in dragging operations. Data in the Clipboard is associated with a name that indicates how it is to be used. You implement data-transfer operations with the Clipboard using Core Foundation Pasteboard Services or the Cocoa NSPasteboard class. See also "pasteboard".

Cocoa An advanced object-oriented development platform in Mac OS X. Cocoa is a set of frameworks used for the rapid development of full-featured applications in the Objective-C language. It is based on the integration of OpenStep, Apple technologies, and Java.

code fragment In the CFM-based architecture, a code fragment is the basic unit for executable code and its static data. All fragments share fundamental properties such as the basic structure and the method of addressing code and data. A fragment can easily access code or data contained in another fragment. In addition, fragments that export items can be shared among multiple clients. A code fragment is structured according to the Preferred Executable Format (PEF).

ColorSync An industry-standard architecture for reliably reproducing color images on various devices (such as scanners, video displays, and printers) and operating systems.

compositing A method of overlaying separately rendered images into a final image. It encompasses simple copying as well as more sophisticated operations that take advantage of transparency.

concrete type Defines, in information property lists, specific characteristics of a type of document, such as extensions and HFS+ type and creator codes. Each concrete type has corresponding abstract types. See also "abstract type".

cooperative multitasking A multitasking environment in which a running program can receive processing time only if other programs allow it; each application must give up control of the processor "cooperatively" in order to allow others to run. Mac OS 9 is a cooperative multitasking environment. See also "preemptive multitasking".

CUPS The Common UNIX Printing System; an open source architecture commonly used by the UNIX community to implement printing.

daemon A process that handles periodic service requests or forwards a request to another process for handling. Daemons run continuously, usually in the background, waking only to handle their designated requests. For example, the `httpd` daemon responds to HTTP requests for web information.

Darwin Another name for the Mac OS X core operating system. The Darwin kernel is equivalent to the Mac OS X kernel plus the BSD libraries and commands essential to the BSD Commands environment. Darwin is an open source technology.

demand paging An operating system facility that causes pages of data to be read from disk into physical memory only as they are needed.

device driver A component of an operating system that deals with getting data to and from a device, as well as the control of that device.

domain An area of the file system reserved for software, documents, and resources and limiting the accessibility of those items. A domain is segregated from other domains. There are four domains: user, local, network, and system.

DVD An optical storage medium that provides greater capacity and bandwidth than CD-ROM; DVDs are frequently used for multimedia as well as data storage.

dyld See "dynamic link editor".

dynamic link editor The library manager for code in the Mach-O executable format. The dynamic link editor is a dynamic library that "lives" in all Mach-O programs on the system. See also "CFM"; "Mach-O".

dynamic linking The binding of modules, as a program executes, by the dynamic link editor. Usually the dynamic link editor binds modules into a program lazily (that is, as they are used). Thus modules not actually used during execution are never bound into the program.

dynamic shared library A library whose code can be shared by multiple, concurrently running programs. Programs share exactly one physical copy of the library code and do not require their own copies of that code. With dynamic shared libraries, a program not only attempts to resolve all undefined symbols at runtime, but attempts to do so only when those symbols are referenced during program execution.

encryption The conversion of data into a form, called ciphertext, that cannot be easily understood by unauthorized people. The complementary process, decryption, converts encrypted data back into its original form.

Ethernet A high-speed local area network technology.

exception An interruption to the normal flow of program control that occurs when an error or other special condition is detected during execution. An exception transfers control from the code generating the exception to another piece of code, generally a routine called an exception handler.

fault In the virtual-memory system, faults are the mechanism for initiating page-in activity. They are interrupts that occur when code tries to access data at a virtual address that is not mapped to physical memory. Soft faults happen when the referenced page is resident in physical memory but is unmapped. Hard (or page) faults occur when the page has been swapped out to backing store. See also "page"; "virtual memory".

file package A directory that the Finder presents to users as if it were a file. In other words, the Finder hides the contents of the directory from users. This opacity discourages users from inadvertently (or intentionally) altering the contents of the directory.

file system A part of the kernel environment that manages the reading and writing of data on mounted storage devices of a certain volume format. A file system can also refer to the logical organization of files used for storing and retrieving them. File systems specify conventions for naming files, storing data in files, and specifying locations of files. See also "volume format".

filters The simplest unit used to modify image data from Core Image. One or more filters may be packaged into an "image units" and loaded into a program using the Core image framework. Filters can contain executable or nonexecutable code.

firewall Software (or a computer running such software) that prevents unauthorized access to a network by users outside the network. (A physical firewall prevents the spread of fire between two physical locations; the software analog prevents the unauthorized spread of data.)

fork (1) A stream of data that can be opened and accessed individually under a common filename. The Mac OS Standard and Extended file systems store a separate data fork and resource fork as part of every file; data in each fork can be accessed and manipulated independently of the other. (2) In BSD, `fork` is a system call that creates a new process.

framebuffer A highly accessible part of video RAM (random-access memory) that continuously updates and refreshes the data sent to the devices that display images onscreen.

framework A type of bundle that packages a dynamic shared library with the resources that the library requires, including header files and reference documentation.

HFS Hierarchical File System. The Mac OS Standard file-system format, used to represent a collection of files as a hierarchy of directories (folders), each of which may contain either files or other folders. HFS is a two-fork volume format.

HFS+ Hierarchical File System Plus. The Mac OS Extended file-system format. This format was introduced as part of Mac OS 8.1, adding support for filenames longer than 31 characters, Unicode representation of file and directory names, and efficient operation on very large disks. HFS+ is a multiple-fork volume format.

HIToolbox Human Interface Toolbox. A collection of procedural APIs that apply an object-oriented model to windows, controls, and menus for Carbon applications. The HI Toolbox supplements older Macintosh Toolbox managers such as the Control Manager, Dialog Manager, Menu Manager, and Window Manager from Mac OS 9.

host The computer that is running (is host to) a particular program; used to refer to a computer on a network.

image units A plug-in bundle for use with the Core Image framework. Image units contain one or more "filters" for manipulating image data.

information property list A property list that contains essential configuration information for bundles. A file named `Info.plist` (or a

platform-specific variant of that filename) contains the information property list and is packaged inside the bundle.

inheritance In object-oriented programming, the ability of a superclass to pass its characteristics (methods and instance variables) on to its subclasses.

instance In object-oriented languages such as Java and Objective-C, an object that belongs to (is a member of) a particular class. Instances are created at runtime according to the specification in the class definition.

internationalization The design or modification of a software product, including its online help and documentation, to facilitate localization. Internationalization of software typically involves writing or modifying code to make use of locale-aware operating-system services for appropriate localized text input, display, formatting, and manipulation. See also "localization".

interprocess communication (IPC) A set of programming interfaces that enables a process to communicate data or information to another process. Mechanisms for IPC exist in the different layers of the system, from Mach messaging in the kernel to distributed notifications and Apple events in the application environments. Each IPC mechanism has its own advantages and limitations, so it is not unusual for a program to use multiple IPC mechanisms. Other IPC mechanisms include pipes, named pipes, signals, message queueing, semaphores, shared memory, sockets, the Clipboard, and application services.

I/O Kit A collection of frameworks, libraries, tools, and other resources for creating device drivers in Mac OS X. The I/O Kit framework uses a restricted form of C++ to provide default behavior and an object-oriented programming model for creating custom drivers.

iSync A tool for synchronizing address book information.

kernel The complete Mac OS X core operating-system environment, which includes Mach, BSD, the I/O Kit, file systems, and networking components. Also called the kernel environment.

key An arbitrary value (usually a string) used to locate a piece of data in a data structure such as a dictionary.

localization The adaptation of a software product, including its online help and documentation, for use in one or more regions of the world, in addition to the region for which the original product was created. Localization of software can include translation of user interface text, resizing of text-related graphical elements, and replacement or modification of user interface images and sound. See also "internationalization".

lock A data structure used to synchronize access to a shared resource. The most common use for a lock is in multithreaded programs where multiple threads need access to global data. Only one thread can hold the lock at a time; this thread is the only one that can modify the data during this period.

manager In Carbon, a library or set of related libraries that define a programming interface.

Mach The lowest level of the Mac OS X kernel environment. Mach provides such basic services and abstractions as threads, tasks, ports, interprocess communication (IPC), scheduling, physical and virtual address space management, virtual memory, and timers.

Mach-O Executable format of Mach object files. See also "PEF".

main thread By default, a process has one thread, the main thread. If a process has multiple threads, the main thread is the first thread in the process. A user process can use the POSIX threading API (Pthread) to create other user threads.

major version A framework version specifier designating a framework that is incompatible with programs linked with a previous version of the framework's dynamic shared library.

makefile A specification file used by a build tool to create an executable version of an application. A makefile details the files, dependencies, and rules by which the application is built.

memory-mapped file A file whose contents are mapped into memory. The virtual-memory system transfers portions of these contents from the file to physical memory in response to page faults. Thus, the disk file serves as backing store for the code or data not immediately needed in physical memory.

memory protection A system of memory management in which programs are prevented from being able to modify or corrupt the memory partition of another program. Mac OS 9 does not have memory protection; Mac OS X does.

method In object-oriented programming, a procedure that can be executed by an object.

minor version A framework version specifier designating a framework that is compatible with programs linked with later builds of the framework within the same major version.

multicast A process in which a single network packet may be addressed to multiple recipients. Multicast is used, for example, in streaming video, in which many megabytes of data are sent over the network.

multihoming The ability to have multiple network addresses in one computer. For example, multihoming might be used to create a system in which one address is used to talk to hosts outside a firewall and the other to talk to hosts inside; the operating system provides facilities for passing information between the two.

multitasking The concurrent execution of multiple programs. Mac OS X uses preemptive multitasking, whereas Mac OS 9 uses cooperative multitasking.

network A group of hosts that can directly communicate with each other.

nib file A file containing resource data generated by the Interface Builder application.

nonretained window A window without an offscreen buffer for screen pixel values.

notification Generally, a programmatic mechanism for alerting interested recipients (or "observers") that some event has occurred during program execution. The observers can be users, other processes, or even the same process that originates the notification. In Mac OS X, the term "notification" is used to identify specific mechanisms that are variations of the basic meaning. In the kernel environment, "notification" is sometimes used to identify a message sent via IPC from kernel space to user space; an example of this is an IPC notification sent from a device driver to the window server's event queue. Distributed notifications provide a way for a process to broadcast an alert (along with additional data) to any other process that makes itself an observer of that notification. Finally, the Notification Manager (a Carbon manager) lets background programs notify users—through blinking icons in the menu bar, by sounds, or by dialogs—that their intercession is required.

NFS Network File System. An NFS file server allows users on the network to share files on other hosts as if they were on their own local disks.

object A programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data. Objects are the principal building blocks of object-oriented programs.

object file A file containing executable code and data. Object files in the Mach-O executable format take the suffix `.o` and are the product of compilation using the GNU compiler (`gcc`). Multiple object files are typically linked together along with required frameworks to create a program. See also "code fragment"; "dynamic linking".

object wrapper Code that defines an object-based interface for a set of procedural interfaces. Some Cocoa objects wrap Carbon interfaces to provide parallel functionality between Cocoa and Carbon applications.

Objective-C An object-oriented programming language based on standard C and a runtime system that implements the dynamic functions of

the language. Objective-C's few extensions to the C language are mostly based on Smalltalk, one of the first object-oriented programming languages. Objective-C is available in the Cocoa application environment.

opaque type In Core Foundation and Carbon, an aggregate data type plus a suite of functions that operate on instances of that type. The individual fields of an initialized opaque type are hidden from clients, but the type's functions offer access to most values of these fields. An opaque type is roughly equivalent to a class in object-oriented programming.

OpenGL The Open Graphics Language; an industry-wide standard for developing portable 2D and 3D graphics applications. OpenGL consists of an API and libraries that developers use to render content in their applications.

open source A definition of software that includes freely available access to source code, redistribution, modification, and derived works. The full definition is available at `www.opensource.org`.

Open Transport Open Transport is a communications architecture for implementing network protocols and other communication features on computers running the Mac OS. Open Transport provides a set of programming interfaces that supports, among other things, both the AppleTalk and TCP/IP protocols.

package In Java, a way of storing, organizing, and categorizing related Java class files; typical package names are `java.util` and `com.apple.cocoa.foundation`. See also "application packaging".

page The smallest unit, measured in bytes, of information that the virtual memory system can transfer between physical memory and backing store. As a verb, page refers to transferring pages between physical memory and backing store.

pasteboard Another name for the "Clipboard".

PEF Preferred Executable Format. An executable format understood by the Code Fragment Manager. See also "Mach-O".

permissions In BSD, a set of attributes governing who can read, write, and execute resources in the file system. The output of the `ls -l` command represents permissions as a nine-position code segmented into three binary three-character subcodes; the first subcode gives the permissions for the owner of the file, the second for the group that the file belongs to, and the last for everyone else. For example, `-rwxr-xr--` means that the owner of the file has read, write, execute permissions (rwx); the group has read and execute permissions (r-x); everyone else has only read permissions. (The leftmost position indicates whether this is a regular file (-), a directory (d), a symbolic link (l), or a special pseudo-file device.) The execute bit has a different semantic for directories, meaning they can be searched.

physical address An address to which a hardware device, such as a memory chip, can directly respond. Programs, including the Mach kernel, use virtual addresses that are translated to physical addresses by mapping hardware controlled by the Mach kernel.

physical memory Electronic circuitry contained in random-access memory (RAM) chips, used to temporarily hold information at execution time. Addresses in a process's virtual memory are mapped to addresses in physical memory. See also "virtual memory".

pixel The basic logical unit of programmable color on a computer display or in a computer image. The physical size of a pixel depends on the resolution of the display screen.

plug-in An external module of code and data separate from a host (such as an application, operating system, or other plug-in) that, by conforming to an interface defined by the host, can add features to the host without needing access to the source code of the host. Plug-ins are types of loadable bundles. They are implemented with Core Foundation Plug-in Services.

port (1) In Mach, a secure unidirectional channel for communication between tasks running on a single system. (2) In IP transport protocols, an integer identifier used to select a receiver for an incoming packet or to specify the sender of an outgoing packet.

POSIX The Portable Operating System Interface. An operating-system interface standardization effort supported by ISO/IEC, IEEE, and The Open Group.

PostScript A language that describes the appearance (text and graphics) of a printed page. PostScript is an industry standard for printing and imaging. Many printers contain or can be loaded with PostScript software. PostScript handles industry-standard, scalable typefaces in the Type 1 and TrueType formats. PostScript is an output format of Quartz.

preemption The act of interrupting a currently running task in order to give time to another task.

preemptive multitasking A type of multitasking in which the operating system can interrupt a currently running task in order to run another task, as needed. See also "cooperative multitasking".

process A BSD abstraction for a running program. A process's resources include a virtual address space, threads, and file descriptors. In Mac OS X, a process is based on one Mach task and one or more Mach threads.

property list A structured, textual representation of data that uses the Extensible Markup Language (XML) as the structuring medium. Elements of a property list represent data of certain types, such as arrays, dictionaries, and strings.

Pthreads The POSIX Threads package (BSD).

Quartz The native 2D rendering API for Mac OS X. Quartz contains programmatic interfaces that provide high-quality graphics, compositing, translucency, and other effects for rendered content. Quartz is included as part of the Application Services umbrella framework.

Quartz Extreme A technology integrated into the lower layers of Quartz that enables many graphics operations to be offloaded to hardware. This offloading of work to the graphics processor unit (GPU) provides tremendous acceleration for graphics-intensive applications. This technology is enabled automatically by Quartz and OpenGL on supported hardware.

QuickTime Apple's multimedia authoring and rendering technology. QuickTime lets you import and export media files, create new audio and video content, modify existing content, and play back content.

RAM Random-access memory. Memory that a microprocessor can either read or write to.

raster graphics Digital images created or captured (for example, by scanning in a photo) as a set of samples of a given space. A raster is a grid of x-axis (horizontal) and y-axis (vertical) coordinates on a display space. (Three-dimensional images also have a z coordinate.) A raster image identifies the monochrome or color value with which to illuminate each of these coordinates. The raster image is sometimes referred to as a bitmap because it contains information that is directly mapped to the display grid. A raster image is usually difficult to modify without loss of information. Examples of raster-image file types are BMP, TIFF, GIF, and JPEG files. See also "vector graphics".

real time In reference to operating systems, a guarantee of a certain capability within a specified time constraint, thus permitting predictable, time-critical behavior. If the user defines or initiates an event and the event occurs instantaneously, the computer is said to be operating in real time. Real-time support is especially important for multimedia applications.

reentrant The ability of code to process multiple interleaved requests for service nearly simultaneously. For example, a reentrant function can begin responding to one call, be interrupted by other calls, and complete them all with the same results as if the function had received and executed each call serially.

resolution The number of pixels (individual points of color) contained on a display monitor, expressed in terms of the number of pixels on the horizontal axis and the number on the vertical axis. The sharpness of the image on a display depends on the resolution and the size of the monitor. The same resolution will be sharper on a smaller monitor and gradually lose sharpness on larger monitors because the same number of pixels are being spread out over a larger area.

resource Anything used by executable code, especially by applications. Resources include images, sounds, icons, localized strings, archived user interface objects, and various other things. Mac OS X supports both Resource Manager–style resources and "per-file" resources. Localized and nonlocalized resources are put in specific places within bundles.

retained window A window with an offscreen buffer for screen pixel values. Images are rendered into the buffer for any portions of the window that aren't visible onscreen.

role An identifier of an application's relation to a document type. There are five roles: Editor (reads and modifies), Viewer (can only read), Print (can only print), Shell (provides runtime services), and None (declares information about type). You specify document roles in an application's information property list.

ROM Read-only memory, that is, memory that cannot be written to.

run loop The fundamental mechanism for event monitoring in Mac OS X. A run loop registers input sources such as sockets, Mach ports, and pipes for a thread; it also enables the delivery of events through these sources. In addition to sources, run loops can also register timers and observers. There is exactly one run loop per thread.

runtime The period of time during which a program is being executed, as opposed to compile time or load time. Can also refer to the runtime environment, which designates the set of conventions that arbitrate how software is generated into executable code, how code is mapped into memory, and how functions call one another.

Safari Apple's web browser. Safari is the default web browser that ships with Mac OS X.

scheduling The determination of when each process or task runs, including assignment of start times.

SCSI Small Computer Systems Interface. A standard connector and communications protocol used for connecting devices such as disk drives to computers.

script A series of statements, written in a scripting language such as AppleScript or Perl, that instruct an application or the operating system to perform various operations. Interpreter programs translate scripts.

semaphore A programming technique for coordinating activities in which multiple processes compete for the same kernel resources. Semaphores are commonly used to share a common memory space and to share access to files. Semaphores are one of the techniques for interprocess communication in BSD.

server A process that provides services to other processes (clients) in the same or other computers.

sheet A dialog associated with a specific window. Sheets appear to slide out from underneath the window title and float above the window.

shell An interactive programming language interpreter that runs in a Terminal window. Mac OS X includes several different shells, each with a specialized syntax for executing commands and writing structured programs, called shell scripts.

SMP Symmetric multiprocessing. A feature of an operating system in which two or more processors are managed by one kernel, sharing the same memory and having equal access to I/O devices, and in which any task, including kernel tasks, can run on any processor.

socket (1) In BSD-derived systems, a socket refers to different entities in user and kernel operations. For a user process, a socket is a file descriptor that has been allocated using `socket(2)`. For the kernel, a socket is the data structure that is allocated when the kernel's implementation of the `socket(2)` call is made. (2) In AppleTalk protocols, a socket serves the same purpose as a "port" in IP transport protocols.

spool To send files to a device or program (called a spooler or daemon) that puts them in a queue for later processing. The print spooler controls output of jobs to a printer. Other devices, such as plotters and input devices, can also have spoolers.

subframework A public framework that packages a specific Apple technology and is part of an umbrella framework. Through various mechanisms, Apple prevents or discourages developers from including or directly linking with subframeworks. See also "umbrella framework".

symbolic link A lightweight reference to files and folders in UFS file systems. A symbolic link allows multiple references to files and folders without requiring multiple copies of these items. Symbolic links are fragile because if what they refer to moves somewhere else in the file system, the link breaks. However, they are useful in cases where the location of the referenced file or folder will not change. See also "alias".

system framework A framework developed by Apple and installed in the file-system location for system software.

task A Mach abstraction, consisting of a virtual address space and a port name space. A task itself performs no computation; rather, it is the context in which threads run. See also "thread".

TCP/IP Transmission Control Protocol/Internet Protocol. An industry-standard protocol used to deliver messages between computers over the network. TCP/IP support is included in Mac OS X.

thread In Mach, the unit of CPU utilization. A thread consists of a program counter, a set of registers, and a stack pointer. See also "task".

thread-safe code Code that can be used safely by several threads simultaneously.

timer A kernel resource that triggers an event at a specified interval. The event can occur only once or can be recurring. Timers are one of the input sources for run loops. Timers are also implemented at higher levels of the system, such as CFTimer in Core Foundation and NSTimer in Cocoa.

transformation An alteration to a coordinate system that defines a new coordinate system. Standard transformations include rotation, scaling, and translation. A transformation is represented by a matrix.

UDF Universal Disk Format. The file-system format used in DVD disks.

UFS UNIX file system. An industry-standard file-system format used in UNIX-like operating systems such as BSD. UFS in Mac OS X is a derivative of 4.4BSD UFS. Its disk layout is not compatible with other BSD UFS implementations.

umbrella framework A system framework that includes and links with constituent subframeworks and other public frameworks. An umbrella framework "contains" the system software defining an application environment or a layer of system software. See also "subframework".

Unicode A 16-bit character set that assigns unique character codes to characters in a wide range of languages. In contrast to ASCII, which defines 128 distinct characters typically represented in 8 bits, Unicode comprises 65536 distinct characters that represent the unique characters used in many languages.

vector graphics The creation of digital images through a sequence of commands or mathematical statements that place lines and shapes in a two-dimensional or three-dimensional space. One advantage of vector graphics over bitmap graphics (or raster graphics) is that any element of the picture can be changed at any time because each element is stored as an independent object. Another advantage of vector graphics is that the resulting image file is typically smaller than a bitmap file containing the same image. Examples of vector-image file types are PDF, encapsulated PostScript (EPS), and SVG. See also "raster graphics".

versioning With frameworks, schemes to implement backward and forward compatibility of frameworks. Versioning information is written into a framework's dynamic shared library and is also reflected in the internal structure of a framework. See also "major version"; "minor version".

VFS Virtual File System. A set of standard internal file-system interfaces and utilities that facilitate support for additional file systems. VFS provides an infrastructure for file systems built into the kernel.

virtual address A memory address that is usable by software. Each task has its own range of virtual addresses, which begins at address zero. The Mach operating system makes the CPU hardware map these addresses onto physical memory only when necessary, using disk memory at other times. See also "physical address".

virtual machine (VM) A simulated computer in that it runs on a host computer but behaves as if it were a separate computer. The Java virtual machine works as a self-contained operating environment to run Java applications and applets.

virtual memory The use of a disk partition or a file on disk to provide the facilities usually provided by RAM. The virtual-memory manager in Mac OS X provides 32-bit (minimum) protected address space for each task and facilitates efficient sharing of that address space.

VoiceOver A spoken user interface technology for visually impaired users.

volume A storage device or a portion of a storage device that is formatted to contain folders and files of a particular file system. A hard disk, for example, may be divided into several volumes (also known as partitions).

volume format The structure of file and folder (directory) information on a hard disk, a partition of a hard disk, a CD-ROM, or some other volume mounted on a computer system. Volume formats can specify such things as multiple forks (HFS and HFS+), symbolic and hard links (UFS), case sensitivity of filenames, and maximum length of filenames. See also "file system".

widget An HTML-based program that runs in the Dashboard layer of the system.

window server A systemwide process that is responsible for rudimentary screen displays, window compositing and management, event routing, and cursor management. It coordinates low-level windowing behavior and enforces a fundamental uniformity in what appears on the screen.

Xcode An integrated development environment for creating Mac OS X software. Xcode incorporates compiler, debugger, linker, and text editing tools into a single package to streamline the development process.

Index

Symbols

Numerals

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z