Inside Mac OS X

System Overview

February 2003

Contents

Figures, Listings, and Tables

C H A P T E R 1

About This Book

Apple Computer's Mac OS X operating system is the most revolutionary operating system to hit the computer scene in many years. With Mac OS X, Apple has reasserted its leadership not only in operating systems but in the technological sophistication and design sensibility that are the hallmarks of the company. While preserving the famed ease-of-use and personality of its predecessors, Mac OS X is an industrial-strength, modern operating system engineered for reliability, stability, scalability, and performance. As such, it lays the foundation for another decade of innovation.

This book introduces software developers to Mac OS X. It describes the operating system's features and architecture. And it explains some of the concepts and conventions of Mac OS X that are of interest to those developing software for the platform.

Why Read This Book

Inside Mac OS X: System Overview is intended for anyone who wants to develop software for Mac OS X. But it is also a resource for people who are just curious about Mac OS X as a development and deployment platform. Whether your background is software development for Mac OS 9, UNIX, Windows, Java, or any other platform, you are likely to find something of value in this book.

This book describes the Mac OS X operating system from both a functional and architectural perspective and explains some of the concepts, services, and conventions common to the three primary development environments: Carbon, Cocoa, and Java. The book attempts to be "API-agnostic," avoiding as much as possible details specific to a programming interface or application environment.

The book has the following chapters:

- **System Technologies**. Describes the user experience and summarizes the features and capabilities of the operating system, including the core operating system called Darwin, the graphics and windowing system, and supported networking services and protocols.
- **System Architecture**. Provides a high-level discussion of the design of Mac OS X, describing the various layers of system software. Also explains how events are handled and discusses some general programming issues.
- **Booting and Logging In**. Describes the sequence of actions that occur when a Mac OS X system boots and when users log into the system. Also shows how you can customize the booting and login sequences.
- **Bundles**. Describes bundles, the basic packaging model for software on Mac OS X.
- **Application Packaging**. Offers details of application bundles and how they package their various resources.
- **Frameworks**. Describes frameworks, another type of bundle, which are used to package dynamic shared libraries and their supporting resources.
- **Umbrella Frameworks**. Provides information about umbrella frameworks, the primary model for packaging Apple-provided frameworks.
- **The File System**. Discusses topics related to the file system, such as the standard directory layout, resource forks, and differences between the major volume formats.
- **The Finder**. Describes the interfaces between the Finder and applications and explains how the Finder handles various tasks, such as determining application ownership of documents and copying files between volumes of different formats.
- **Software Configuration**. Describes the basic mechanisms for configuring applications and other bundles and for handling user preferences.
- **Internationalization**. Explains how to prepare your application for localization and how to enable it to display multiscript text.

- **Installation and Integration**. Summarizes some key guidelines concerning documents and applications, discusses some integration issues, and provides an overview of installation options and techniques.
- **Issues and Options With Multiple Environments**. Discusses some of the programming issues arising from multiple application environments and layered architecture in Mac OS X.
- **Information Property List Keys**. Lists the keys available for use in the information property list of a bundle.

Further Investigations

This book serves as a starting point. It defines the broad conceptual terrain of Mac OS X, and you must go elsewhere to learn about details mentioned or only suggested by the "map." For example, for information about creating a bundle, you should see the documentation for Apple's developer tools.

This section lists sources of Mac OS X information for software developers. It is by no means an exhaustive list, and Apple's contribution to this list will grow.

Installed Developer Documentation

When you install the Developer package of Mac OS X, the Installer application puts developer documentation into four locations:

- Frameworks. Information that is inextricably associated with a framework is usually installed in a localized subdirectory of the framework. This method of packaging ensures that the documentation moves with the framework when and if it moves (or is copied) to another location. It also makes it possible to have localized versions of the documentation (although English currently is the only supported localization).
- Development applications. Help information for all developer applications is installed in the `/Developer/Documentation` folder. When users request it from the Help menu, most applications launch Help Viewer to display it. Project Builder now displays help from directly within the Project Builder interface.

- Example code. A variety of sample programs are installed in `/Developer/Examples` showing you how to perform common tasks using the primary Mac OS X application environments—Carbon, Cocoa, and Java.
- All information that is not specific to frameworks or development applications is installed in `/Developer/Documentation`. The Installer also creates symbolic links to the framework documentation in this location.

You can access Apple's developer documentation from Help Viewer and from Project Builder. Both applications present the currently installed documentation in a browser-like window for you to navigate. You navigate the documentation using this window, clicking links to move from page to page. However, clicking the link of an external URL opens that URL in your preferred Web browser.

To view documentation using Help Viewer, choose Mac Help from the Finder Help menu and select the topic you want to view. The Developer Help Center is in the Help Viewer drawer. This drawer contains the "books" that are currently installed. You can search for topics in Help Viewer using the controls in the window toolbar. The scope of a search in Help Viewer is determined by your current location within the set of books. If you are browsing through a particular book—say, Core Technologies—and you search for a term or API symbol, the Help Viewer first looks at the Apple Help index for that book. If it cannot find the term or symbol, it searches all books in the Developer Help Center.

You can also access the Developer Help Center directly from Project Builder. The Help menu in Project Builder contains commands for viewing the Developer Help Center top-level page or for viewing specific areas, such as Cocoa, Carbon, and the Release Notes. Selecting one of these commands displays the corresponding help within a Project Builder window.

Other Apple Publications

Apple is planning a series of Inside Mac OS X books. This book, *Inside Mac OS X: System Overview*, is the first of that series. At the time of publication, Apple has a publish-on-demand arrangement with Vervanté. Through this arrangement you can obtain books in the Inside Mac OS X series as they become available.

To obtain your printed copy of an Inside Mac OS X book, use your Web browser to access the page at http://www.vervante.com/apple. Then follow the directions.

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 a variant of UNIX. Several excellent books on BSD and UNIX are available in most technical bookstores (or bookstores with technical sections).

You can also use the World Wide Web as a resource for information on BSD. Several organizations, which make available their own free versions of BSD, maintain websites with manuals, FAQs, and other sources of information:

- The Darwin project, http://developer.apple.com/darwin
- The FreeBSD project, http://www.freebsd.org
- The NetBSD project, http://www.netbsd.org
- the OpenBSD project, http://www.openbsd.org

See the bibliography in *Inside Mac OS X: Kernel Programming* for more references.

Other Information on the Web

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

- Apple product information (http://www.apple.com/macosx). Provides general information on Mac OS X.
- Apple Developer Connection—Developer Documentation (http://developer.apple.com/techpubs). Features the same documentation that is installed with Mac OS X and also includes updated and new documentation. Also includes legacy documentation.
- Apple Knowledge Base (http://kbase.info.apple.com). Contains technical articles, tutorials, FAQs, technical notes, and other information.
- Apple Developer Connection—Mac OS X (http://developer.apple.com/macosx). Offers SDKs, release notes, product notes and news, and other resources and information related to Mac OS X.

C H A P T E R 2

System Technologies

Mac OS X is both a radical departure from previous Macintosh operating systems and a natural evolution from them. It carries on the Macintosh tradition of ease-of-use, but more than ever it is designed not only to be easy to use but a pleasure to use.

This next-generation operating system is a synthesis of technologies; some are new and most are standard in the computer industry. It is firmly fixed on the solid foundation of a modern core operating system, bringing benefits such as protected memory and preemptive multitasking to Macintosh computing. Mac OS X has a sparkling user interface capable of visual effects such as translucence and drop shadows. These effects, as well as the sharpest graphics ever seen on a personal computer, are made possible by a graphics technology that Apple developed specifically for Mac OS X.

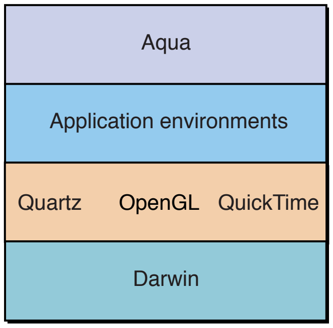
But Mac OS X is more than a sophisticated core and a pretty face. With its multiple application environments, virtually all Macintosh applications can run on it. And with its support for many networking protocols and services, Mac OS X is the ultimate platform for using and enjoying the Internet. It also offers a high degree of interoperability with other operating systems because of its multiple volume formats and its conformance with established and evolving standards.

From a functional perspective, these are the most important components of Mac OS X:

- Aqua, the human-interface design behind the user's experience
- the application environments Carbon, Cocoa, Java, and Classic
- the windowing and graphics system, as implemented by Quartz (and which includes support for QuickTime and OpenGL)
- Darwin, the advanced, UNIX-based core of the operating system

Figure 2-1 depicts the general dependencies between these components. The rest of this chapter describes what these and other technologies of Mac OS X have to offer.

Figure 2-1 A functional view of Mac OS X



The User Experience

The user environment for Mac OS X is similar to that of earlier versions of the Mac OS. But it is also different in important ways. There are differences in the design of the user interface, in the infrastructure for localizing the interface, and in the way to add application features. Mechanisms for exporting and accessing the services of other applications have been enhanced.

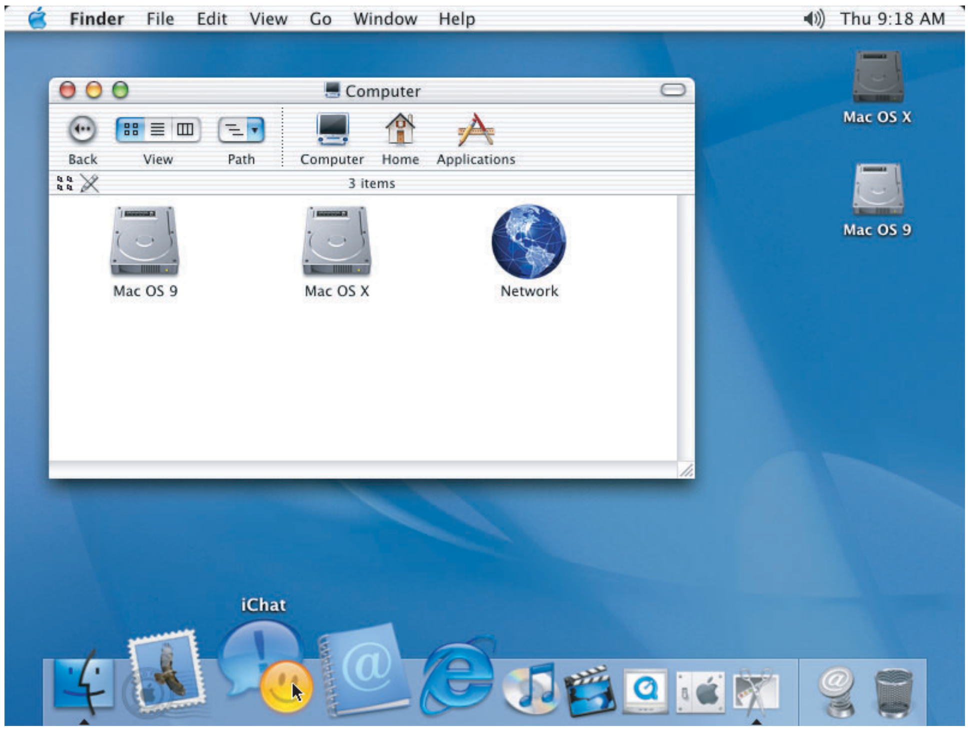
And, of course, the user experience draws from the benefits obtained through the core of the operating system (see "Darwin" (page 33)). A Macintosh computer remains stable even when an application crashes, and no single application or task can monopolize processing resources; applications can execute concurrently.

This section describes the experience that Mac OS X offers to users and the features and applications that make the experience a productive and enjoyable one.

Aqua

When Apple designed Aqua, the graphical user interface for Mac OS X, it had one goal in mind: to create a modern operating system that is not only easy to use, but is more appealing than any Mac OS you've ever seen (see Figure 2-2 for a screen shot). As "aqua" suggests, the properties of water infuse the lucid appearance of Mac OS X. Aqua brings a computer to life with color, depth, clarity, translucence, and motion. Buttons look like polished blue gems, active buttons pulse, windows have drop shadows to give them depth, minimized windows swoop into their Dock icon like a genie into its bottle.

Figure 2-2 The Aqua user interface



One striking characteristic of Aqua is its icons. In earlier operating systems, icon sizes were constrained by the limitations of screen resolution. With today's dramatically improved display sizes and resolution levels, Aqua sheds these constraints. It offers richly colored and photo-quality icons that are adjustable up to 128-by-128 pixels. Its icons are more legible, enabling such features as in-place document previews.

Aqua also improves the user's experience by better management of screen real estate. Operating systems are noted for cluttering up screens by spawning window after window, especially when there are deeply structured file systems and multiple control panels. Mac OS X eliminates the problem of proliferating windows by focusing the activities of an application in a single window.

A prime example of this new approach is how Mac OS X handles common application operations such as opening or printing documents. Time was, when the operating system presented a dialog to print or save a document, you had to know which document the dialog was for, even though you might have many documents open at a time. Mac OS X introduces a new type of dialog, called a *sheet*, that attaches to a document and makes their relationship clear. Sheets appear to slide out from underneath the window title, and their translucent quality makes them look as though they're floating above the document. Sheets are not modal, and thus permit the user to continue with other operations while the sheet is displayed.

The Mac OS has long been admired for its ease of use. Aqua incorporates many of the user-interface qualities and characteristics Macintosh users expect in their computers. Ease of use is factored into just about every feature and capability in the system.

Many of the effects of Aqua are made possible by Quartz, the 2D graphics and windowing technology developed by Apple. See "Quartz" (page 38) for more on this technology. For more information on the Aqua interface, see *Inside Mac OS X: Aqua Human Interface Guidelines*.

The Finder

A big part of the Aqua experience for users is the design of the desktop and the Finder, a system application that acts as the primary interface for file-system interaction. Users are likely to notice two major innovations in this area: the Dock and the way the Finder displays the elements of the file system.

The Dock reduces desktop clutter. This area of the screen holds just about anything you want to keep handy for instant access: folders, applications, documents, storage devices, minimized windows, QuickTime movies, links to websites. An icon identifies each item stored in the Dock; these icons often provide useful feedback about what they represent. For example, the icon for Mail tells you if you have any new messages waiting to be read. If you store an image, the Dock shows it in preview mode, so you can tell what it is without opening it. And because you can minimize running applications into the Dock, a quick look at the bottom of the screen tells you what applications you're currently running. To switch between tasks, simply click the application or document icon you want to start using, and it becomes the new active task. If you don't know what an icon represents, you can move your mouse pointer over it and the title of the document, folder, or application appears.

The Dock holds as many things as you want to keep there. As you add items, the Dock expands until it reaches the edge of the screen. Once it reaches that point, the icons in the Dock shrink proportionately to accommodate additional items. To make the smaller icons more legible, however, Mac OS X includes a feature called magnification: Just pass the cursor over the icons, and they magnify to your preset level.

The Mac OS X Finder has a simple navigation interface that can be contained within a single window. Intuitive controls in a configurable toolbar instantly transport you to the most frequently accessed areas on your computer: your home directory, your applications, your documents, even the people with whom you often communicate. The items that the Finder displays are not only folders, applications, and documents, but other commonly accessed items such as mounted network volumes, external storage devices, CD-ROMs, and digital cameras.

In addition to the icon and list views, Macintosh users are familiar with, each Finder window can be set to the viewing mode called column view. This mode is ideal for navigating deep file systems; clicking a folder displays the contents of that folder in the next column to the right. Column view also maintains a history of your navigation forays so you can always find your way back.

When you double-click Finder items in icon or list view, the Finder by default does not bring up a separate window. Instead, the Finder replaces the old folder view within the single Finder window. (You can change this default behavior, however.) By focusing the file system into a single window view, the Finder reduces the proliferation of windows, a key design goal. Despite this default behavior, nothing prevents you from opening as many Finder windows as you wish.

In Mac OS 9, the Finder identified files using the file type and creator codes stored in the file. The Finder in Mac OS X is capable of recognizing files by their filename extension as well as by their type and creator codes. Since many Mac OS 9 users are used to files without file extensions, the Finder offers a way to hide those extensions globally through the Finder preferences and on a per-file basis through the Show Info window. For more information about files and filename extensions, see "How the File System Is Organized" (page 163).

Application Support

Part of the Mac OS X user experience is the seamless interaction among different components of the operating system. From BSD to QuickTime, Mac OS X consists of technologies with widely different histories and based on different standards and conventions. A single Mac OS X system hosts volumes of different formats, supports different network file-sharing protocols, and can run applications based on radically different APIs.

Mac OS X provides an easy transition for users and developers. To this end, Mac OS X supports four application environments, each intended for a particular type of application:

- The Classic environment can run most Mac OS 9 applications. Because Classic is a compatibility environment, it does not support some Mac OS X features, such as Aqua or core architectural enhancements provided by Darwin.
- The Carbon environment runs all Mac OS 9 applications whose code has been optimized for Mac OS X. By converting their code to use the Carbon APIs, application developers can ensure that applications take advantage of protected memory, preemptive multitasking, and other features of Darwin.
- The Cocoa environment offers an advanced object-oriented framework for creating the best next-generation applications.
- The Java environment runs 100% Pure Java and mixed-API Java applications and applets.

Mac OS X makes it possible to copy (or cut) almost any piece of data and paste it into an application executing in another environment. It also enables dragging of Finder objects (and the data they represent) between most environments. Mac OS X performs all necessary conversions when, for example, a file stored on a Mac OS Extended (HFS+) volume is copied to a UFS volume.

Mac OS X application packaging makes it possible for multiple application executables to coexist in a directory that, to a user, looks and behaves like a double-clickable file. Included in this directory are the resources the executables need, such as images, sounds, localized strings, plug-ins, and private and shared libraries. With this scheme, you can install the same application package on a Mac OS X and a Mac OS 9 system and users can launch and use the application. Because an application package contains everything an application needs to execute on more than one system, certain advanced features become easier to realize, such as remotely executing an application on a server, distributing applications over the Internet, and simplified installation and uninstallation. See the chapter "Application Packaging" (page 131) for more information.

Multiple Users

Users work on a Mac OS X system in a personally customized environment. They can select a desktop pattern, their preferred language, the applications to start up at boot-time, and a number of other preferences. Whenever they log in to their account, all of their choices are restored.

A user's personalized environment is potentially one of many such environments. Other users can log in to the same computer and have an entirely different set of preferences define their computing environment. Mac OS X enforces secure boundaries between one user's data and programs and another's. Each account is password-protected and users cannot execute applications or edit or even read documents in another user's folder without the owner's permission. The system gives each user's folder (and all it contains) a default set of permissions that the user can thereafter change to restrict access or grant greater access to other users.

More powerful than this single (local) machine/multiple users model is the multiple machines/multiple users model—in other words, network accounts, which Mac OS X makes possible through its NetInfo network management system. People can use any Mac OS X system connected to their NetInfo network—which can be a home computer, a portable computer, or a system in a friend's house—to log in to their account on a remote server. When logged in, they can work in an environment that is exactly like it was when they last logged out, regardless of which machine they last used to log in. And if a site is properly administered, their information on that server is just as secure as any locally maintained data, perhaps more secure if files on the server are backed up regularly.

The preferences system in Mac OS X is flexible enough to support any combination of remote and local access. With it, users and administrators can specify sets of preferences on per-user, per-machine, and per-application bases.

Internationalization

Mac OS X makes it easy to internationalize software. And it does so in such a way that a single binary can support localizations for multiple languages and regional dialects. It also lets software developers dynamically add localized resources for new languages or regions.

Mac OS X includes comprehensive technology to handle text systems used around the world. This text system provides Unicode, input methods, and general text handling services. In Mac OS X, most software comes in the form of a bundle, of which an application is just one type (see "Application Support" (page 28)). A bundle is an opaque directory in the file system that contains one or more executables and the resources that go with those executables. One of the primary benefits of bundles is the infrastructure they provide for localizing software. For users, a bundle appears to be a single file object that can be double-clicked or dragged from folder to folder.

Localized resources such as image and strings files, as well as Mac OS 9–style resources (`.rsrc`), can be put in bundle subdirectories whose names reflect a particular language or regional dialect (for example, Canadian French). A properly constructed Mac OS X application (or plug-in or shared library) does not hardwire paths to the resource files in these directories. Instead, when the application needs a resource, it uses a special system routine to obtain the localization that best matches the user's language preferences.

See the chapters "Internationalization" (page 209), "Application Packaging" (page 131), and "Bundles" (page 115)) for further information.

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 designed to accommodate users with special needs. It also provides software developers with the functions they need to support accessibility in their own applications.

Mac OS X offers the following accessibility support:

- Zoom features let users increase the size of onscreen elements.
- Sticky keys let users press keys sequentially instead of simultaneously, so that they can use 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.

In addition to its built-in support, software developers can use Carbon and Cocoa APIs to communicate accessibility information to other applications. Cocoa controls implement the NSAccessibility protocol, which communicates accessibility information to the system. In Carbon there are functions that provide similar support.

For details, see *Inside Mac OS X: Making Your Application Accessible to Users With Disabilities*.

AppleScript

Scripting in Mac OS X, as in Mac OS 9, employs AppleScript as the primary scripting language and Apple events as the communication model. You can program behavior into your applications so they act appropriately upon receiving AppleScript commands. AppleScript is supported in all application environments as well as in the Classic compatibility environment. Users can thus write scripts that link together the services of multiple applications in different environments.

When designing new applications, AppleScript support should be considered early in the process. Consider your application's data structures carefully. Supporting AppleScript requires designing your application's data structures in an efficient way that also makes sense for people writing scripts. It also makes it easier for other applications to communicate with your application quickly and effectively.

AppleScript Studio is a powerful tool for quickly creating native Mac OS X applications that support the Aqua user interface guidelines. It combines features from AppleScript, Project Builder, Interface Builder, and the Cocoa application framework. With AppleScript Studio, you can work in a full-featured development environment to create applications that use AppleScript scripts to control a broad range of Cocoa user-interface objects.

Exported Application Services

Applications running concurrently in a Mac OS X system don't have to run in isolation. Any application can make a service it provides available to other applications, and any application interested in that service can take advantage of it. The Services menu provides an additional way for applications to share resources and capabilities.

Applications do not have to know what services are offered in advance. A user selects a piece of data in an application, such as a string of text, or an image, or an icon representing a folder or file. Then user then selects a command from an application listed in the Services menu and the command is executed on the selection, invoking that second application.

The Services facility often works as though the user copies data from one application, pastes it into another, modifies the data, then copies the result and pastes it back into the original application. For example, a user might select a folder in the Finder and choose a Services option that compresses the folder and puts it into an archive format; the result of this operation is placed back in the same place as the original folder. But the action can be one way as well; for instance, a user might select a name in a word-processing document and choose a Services command that looks up the name using an LDAP server, starts up an email application, and opens a new message window with the found email address after the To: line.

Other Parts of the User Experience

As with prior versions of the Mac OS, the user's experience of Mac OS X begins when the box containing the CD-ROM is opened. Installation is a simple task and a set-up assistant has the user up and running and on the Internet quickly. If users have questions, they can use the Help Viewer to find the answers.

Mac OS X integrates the Internet into everyday computer use. It makes it easy for users to access the Internet and to save the locations of favorite websites for later access. It features Sherlock for searching the Internet or an intranet as well as for searching the local file system (including searching by indexed content). Mac OS X also includes a powerful, yet incredibly easy-to-use, email application based completely on Internet standards.

Darwin

Beneath the appealing, easy-to-use interface of Mac OS X is a rock-solid, UNIX-based foundation that is engineered for stability, reliability, and performance. This foundation is a core operating system commonly known as Darwin, which is an Open Source technology available from `www.apple.com/darwin`. Darwin integrates a number of technologies, most importantly Mach 3.0, operating-system services based on 4.4BSD (Berkeley Software Distribution), 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 the books *Inside Mac OS X: Kernel Programming* and *Inside Mac OS X: I/O Kit Fundamentals*. These books are available on the Apple website: http://developer.apple.com/techpubs/macosx/Darwin.

Mach

Mach is at the heart of Darwin because it performs a number 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 brings many important advantages to Macintosh computing:

- **Protected memory**. The stability of an operating system should not depend on all executing applications being good "citizens" by not writing data to each others's (or the system's) address space; doing so can result in loss or corruption of information and can even precipitate system crashes. Mach ensures that an application cannot write on another application's memory or on 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 hurt the rest of the system. And, perhaps best of all, if an application crashes, it doesn't affect the rest of the system and so you don't need to restart your computer.

- **Preemptive multitasking**. In a modern operating system, 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**. Like other virtual memory systems, 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 is resident 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. In Mac OS X, virtual memory is "on" all the time.
- **Real-time support**. This feature guarantees low-latency access to processor resources for time-sensitive media applications.

Darwin also enables cooperative multitasking and preemptive and cooperative threading.

BSD

Integrated with Mach is a customized version of the BSD operating system (currently 4.4BSD). Darwin's implementation of BSD includes much of the POSIX API and exports it to 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 user IDs and permissions
- threading support (POSIX threads)
- BSD sockets

Device-Driver Support

For development of device drivers, Darwin offers an object-oriented framework called the I/O Kit. The I/O Kit not only facilitates the creation of drivers for Mac OS X, but provides much of the infrastructure those drivers need. It is written in a restricted subset of C++. The framework, which is designed to support a range of device families, is both modular and extensible.

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

- true plug and play
- dynamic device management ("hot plugging")
- power management (both desktops and portables)

For information on creating device drivers, see *Inside Mac OS X: I/O Kit Fundamentals*. For descriptions of the device drivers developed by Apple, see "Advanced Hardware Features" (page 49).

Networking Extensions

Darwin gives kernel developers a new 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 detailed information on developing networking extensions with NKE, see *Inside Mac OS X: Network Kernel Extensions*. For descriptions of the networking services and protocols natively implemented in Darwin, see "Networking and the Internet" (page 45).

File Systems

The file-system component of Darwin is based on extensions to BSD and an enhanced Virtual File System (VFS) design. The file-system component introduces several new general features:

- Permissions on removable media. This feature is based on a globally unique ID registered in a system for each connected removable device (including USB and FireWire devices).
- 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.

Because of its multiple application environments and the various kinds of devices it supports, Mac OS X must be able to handle file data on many standard volume formats. Table 2-1 lists the supported formats.

Table 2-1 Supported local volume formats

Mac OS Extended Format	Also called 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. It is also the standard volume format for Mac OS 9.
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 (as does HFS+) stores resources and data in separate “forks” of a file and makes use of various file attributes, including type and creator codes.
UFS	A “flat” (that is, single-fork) disk volume format, based on the 4.4BSD 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	The Universal Disk Format for DVD volumes.
ISO 9660	The standard format for CD-ROM volumes.

HFS and HFS+ volumes support aliases and UFS volumes support symbolic links (HFS+ and UFS both support hard links). 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. See the chapter "The File System" (page 163) for descriptions of these and other differences.

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 linking together disparate systems, it also supports multiple network file-server protocols. Table 2-2 lists these protocols.

Table 2-2 Supported network file protocols

AFP client	Apple File 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.
Samba	SMB/CIFS, a file-sharing protocol used on Windows and UNIX systems.

Some file-system capabilities extend to all writable volume formats in Mac OS X.

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. Being Open Source technology, Darwin is a key part of that strategy. Apple has released the source code to virtually all of the components of Darwin to the developer community.

The Mac OS X kernel environment is a subset of Darwin. The kernel environment contains everything in Darwin except the BSD libraries and commands that are essential to the BSD Commands environment. For more on the kernel environment, see the book *Inside Mac OS X: Kernel Programming*.

Graphics and Imaging

Mac OS X combines Quartz, QuickTime, and OpenGL—three of the most powerful graphics technologies available—to take the graphics capabilities of the Macintosh beyond anything seen on a desktop operating system. The two-dimensional graphics and imaging capabilities of Mac OS X are based on Quartz, an Apple technology that provides a window server and essential low-level services as well as a graphics rendering library that uses PDF (Portable Document Format) as its internal model. Integrated into this foundation is a printing architecture and other graphics libraries such as QuickDraw and QuickTime.

Quartz

Quartz is a powerful graphics system that delivers a rich imaging model, on-the-fly rendering, anti-aliasing, and compositing of PostScript graphics. Quartz also implements the windowing system for Mac OS X and provides low-level services such as event handling and cursor management. It also offers facilities for rendering and printing that use PDF as an internal model for graphics representation.

Table 2-3 describes some of Quartz's rendering capabilities and other features.

Table 2-3 Quartz graphics capabilities and specifications

Bit depth	A minimum bit depth of 16 bits for typical users. An 8-bit depth in full-screen mode is available for 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 games and other multimedia applications.
Anti-aliasing	All graphics and text are anti-aliased.
Frame buffer access	Includes a mechanism that lets graphics applications (such as games) gain direct access to the video frame buffer.

Table 2-3 Quartz graphics capabilities and specifications

Velocity Engine	Quartz and QuickDraw both take advantage of the Velocity Engine to boost performance.
Quartz Extreme	Quartz Extreme leverages OpenGL for the entire Mac OS X desktop. Graphics calls now render in supported video hardware, freeing up the CPU for other tasks.
2D graphics acceleration	Supports two-dimensional graphics acceleration, improving what is currently available in QuickDraw. (Acceleration is currently limited to system software and Classic applications; other applications must draw into backing store in DRAM.)
ColorSync color management	Quartz uses ColorSync to manage pixel data when drawing data on the screen, respecting ICC profiles, or applying the system's monitor profile as source color space. ColorSync can also be called when printing occurs.

Quartz has two components, Quartz Compositor and Quartz 2D. The first of these, Quartz Compositor, is essentially the window server for the system. The window server provides the fundamental windowing and event-routing services for all application environments. This high-performance server is lightweight in that it performs no rendering itself, yet it provides essential services to all graphics rendering libraries that are clients of it, including Quartz 2D and QuickDraw. Quartz Compositor features such advanced capabilities as device-independent color and pixel depth, layered compositing, and buffered windows for the automatic repair of window damage. Quartz Compositor also includes transparent support for accelerated graphics hardware using Quartz Extreme.

The Quartz 2D component of Quartz is a graphics rendering library for two-dimensional shapes. It is used for screen rendering, PDF generation, print preview, and other services. Quartz 2D uses PDF as an internal model for vector graphics representation. PDF offers several advantages, including good color management, internal compression, and font independence. Quartz 2D uses a coordinate system that is flexible and precise (because it uses floating-point coordinates) and thus permits some degree of device independence.

Quartz 2D enables a number of important features:

- automatic PDF generation and save-as-PDF
- a consistent feature set for all printers

- automatic onscreen preview of graphics
- conversion of PDF data to printer raster data or PostScript
- high-quality screen rendering
- color management through ColorSync

See "The Graphics and Windowing Environment" (page 67) in the chapter "System Architecture" for more information on Quartz.

QuickDraw

Carbon developers should use the Quartz API whenever possible to render content. However, QuickDraw is also available as a legacy technology for the construction, manipulation, and display of two-dimensional graphical shapes, pictures, and text.

QuickDraw provides a facility for code to send QuickDraw imaging instructions through an interface to the Quartz 2D library. This interface gives QuickDraw code access to the PDF-generation, PostScript-generation, text anti-aliasing, and other graphics and imaging capabilities of Quartz.

OpenGL

Mac OS X includes Apple's highly optimized implementation of OpenGL as the system API and library for three-dimensional (3D) graphics. OpenGL is an industry-wide standard for developing portable 3D graphics applications. OpenGL is one of the most widely adopted graphics API standards today, which makes code written to OpenGL portable and the generated visual effects very consistent. 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 Mac OS X version of OpenGL produces consistently high-quality graphical images 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 special effects, and developers can create lists of these commands for repetitive effects. Although OpenGL is largely independent of the windowing characteristics of each operating system, special “glue” routines are implemented to enable OpenGL to work in an operating system’s windowing environment.

QuickTime

Mac OS X comes packaged with the latest version of 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 also allows you to stream digital video where the data stream can be either live or stored. QuickTime is cross-platform technology; besides Mac OS X, it is available in Mac OS 9, Windows 95, Windows 98, Windows NT, and Windows 2000.

QuickTime supports every major file format for images, including PICT, BMP, GIF, JPEG, TIFF, and PNG. It also supports every significant professional file format for video, including AVI, AVR, DV, M-JPEG, MPEG-1, MPEG-2, MPEG-4, AAC, and OpenDML. For Web streaming, it includes support for HTTP as well as RTP and RTSP.

QuickTime streaming allows users to view live and video-on-demand movies using the industry-standard protocols RTP (Real-Time Transport Protocol) and RTSP (Real-Time Streaming Protocol). Users can view streaming live broadcasts, previously recorded movies, or a mixture of both. Broadcasts can be either unicast (one-to-one) or multicast (one-to-many).

Through the QuickTime plug-in, QuickTime’s digital video streaming capability is extended to all popular Web browsers, including Internet Explorer, Netscape Navigator, and America Online browsers. The plug-in supports over thirty different media types and makes it possible to view over 80 percent of all Internet media. The Web streaming capabilities of QuickTime include a Fast Start feature, which presents the first frame of a movie almost immediately and automatically begins playing a movie as it is downloaded. It also features other advanced capabilities, such as movie “hot spots” and automatic Web-page launching.

Printing

The printing system for Mac OS X is based on an architecture completely different from that of earlier versions of the Mac OS. It is a service available for all application environments. Drawing upon the capabilities of Quartz, the printing system delivers a consistent human interface and makes possible shorter development cycles for printer vendors. It allows applications to draw in "virtual pages" and map those pages to physical pages at print time, breaking the connection between the drawing page and the printing page. The printing system also provides applications with a high degree of control over the user-interface elements in printing dialogs. Table 2-4 describes some additional features.

Table 2-4 Features of the Mac OS X printing system

Print Center	Provides a single interface for finding printers, submitting jobs, and managing queues.
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 print set-up dialog.
PostScript printing	Prints to PostScript Level 1, 2, and 3 compatible printers, except in the Classic environment.
Raster printers	Prints to raster printers in all environments, except in the Classic environment.
Print preview	Provides a print preview capability in all environments, except in Classic. The printing system implements this feature by launching a PDF viewing application. This preview is color-managed by ColorSync.
Print spooling	Enables speedy spooling of print jobs.
CUPS	Supports the Common Unix Printing System architecture. CUPS is a standard printing solution for UNIX-based systems.

For more information on the printing features in Mac OS X, see "The Printing System" (page 72).

Apple Type Solution

The Apple Type Solution (ATS) is the engine for the system-wide 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.

Application Technologies

Mac OS X includes several technologies that extend the functionality available to applications.

Application Extensibility

Plug-ins are modules of code and resources that developers and users can dynamically add to an application to extend its capabilities. The host application structures its code so that well-defined areas of functionality can be provided by external plug-ins. The host does not have to be aware of the implementation details of the plug-in. When the application is launched, it uses mechanisms provided by the plug-in architecture to locate its plug-ins and load them. An application can let users add plug-ins at any time while it is running, and it can also give users the means for removing its plug-ins.

Plug-ins offer a range of benefits for both users and developers. Users can customize the features of an application to suit their requirements, and as new or upgraded functionality (as encapsulated by a new or replacement plug-in) becomes available, users can “plug” these features into the application.

By adopting the Mac OS X plug-in model, developers inherit a single, standard plug-in architecture that eliminates the need for custom architectures. Plug-ins permit an incremental but efficient implementation of features, making it possible to create a custom version of an application without changing the original code base. Because they are separate modules, plug-ins help developers to isolate and correct bugs in the software. They also make it possible for third-party developers to add value to an application without the involvement of the original developer.

Important
Developers should always be careful when deciding how to factor their application to support plug-ins. Despite the benefits of plug-ins, loading a large number of plug-ins can incur noticeable performance penalties. If an application relies too heavily on plug-ins, its appearance may seem sluggish to the user.

For details, see the conceptual and reference documentation for Core Foundation Bundle Services and Plug-in Services.

Disc Recording

Mac OS X 10.2 introduces a new framework that gives applications the ability to burn and erase CDs and DVDs. The Disc Recording framework was built to satisfy the simple needs of a general application, making it easy to add basic audio and data burning capabilities to any application. At the same time, the framework is flexible enough to support professional CD and DVD mastering applications.

The Disc Recording framework (`DiscRecording.framework`) minimizes the amount of work your application must perform. Your application is responsible for specifying the content to be burned. Once your application specifies the content, 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 and 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.

Contact Database

Mac OS X 10.2 introduces 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 can use this data as is or extend it to include application-specific information.

The Address Book framework provides a way to access user records and create new ones. Applications that support this framework gain the ability to share contact information with other applications. The API also provides the concept of a “Me” record for the current user. This record contains information about the currently logged in user and can be used by programs such as Web browsers to automatically fill in Web forms with appropriate data.

Networking and the Internet

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 35), facilitates the creation of modules implementing new or existing protocols that can be added to this stack.

Media Types

Mac OS X supports the network media types listed in Table 2-5.

Table 2-5 Network media types

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.

Table 2-5 Network media types

Jumbo Frame	This Ethernet format is a technology that 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.
Serial	Supports modem, DSL, and ISDN capabilities.
Wireless	See "AirPort" (page 50).

Standard Protocols

Mac OS X supports a number of protocols that are standard in the computing industry. Table 2-6 summarizes these protocols.

Table 2-6 Network protocols

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.)
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.
PAP	The Printer Access Protocol is used for spooling print jobs and printing to network printers.
HTTP	The Hypertext Transport Protocol is the standard protocol for transferring Web pages between a Web server and browser.
FTP	The File Transfer Protocol (part of BSD) is a standard means of moving files between computers on TCP/IP networks.
DNS	Domain Name Services is the standard Internet service for mapping host names to IP addresses.

Table 2-6 Network protocols (continued)

SLP	Service Location Protocol is a protocol designed for the automatic discovery of resources (printers, servers, fax machines, and so on) on an IP network.
DHCP and BOOTP	The Dynamic Host Configuration Protocol and the Bootstrap Protocol automate the assignment of IP addresses in a particular network.
LDAP	The Lightweight Directory Access Protocol lets users locate organizations, individuals, and resources such as files and devices in a network, whether on the Internet or on a corporate intranet.
NTP	The Network Time Protocol is used for synchronizing client clocks.
SOAP	The Simple Object Access Protocol is a lightweight protocol for exchanging encapsulated messages over the Web or other networks.
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 2-2 (page 37) 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 Macintosh and can be integrated with other network systems, such as the Internet. Mac OS X includes minimal support for compatibility with legacy AppleTalk environments and solutions.
- Open Transport implements industry-standard communications and networking protocols as part of the I/O system. It helps developers to 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 the new networking technologies provided by Cocoa and Carbon.

Rendezvous

Mac OS X 10.2 and later include support for Rendezvous, Apple's implementation of zero-configuration networking. Rendezvous makes the dynamic discovery of file servers and printers much simpler and truly more plug-and-play. Using Rendezvous, computers can create ad-hoc networks over Ethernet or Airport connections.

Rendezvous delivers the ability to dynamically discover computer services over TCP/IP networks without the need for any complex user configuration of the associated hardware. Rendezvous connects computers or electronic devices by configuring them automatically for a common network. Each device then broadcasts the services it offers to other devices on the network.

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. Mac OS X offers IP routing support for just these occasions. With IP routing, a Mac OS X machine 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.

Personal File and Web Services

Personal Web Sharing, which is also a feature of Mac OS 9, 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.

Advanced Hardware Features

Right out of the box, Mac OS X supplies drivers for most standards-based hard drives and add-on devices in common use today. For example, it provides support and drivers for IDE and SCSI disk drives and supports a wide range of Apple monitors. Mac OS X also includes features such as power management for both desktop and portable systems.

The rest of this section discusses some of the advanced hardware features of Mac OS X. For hardware-related information in this book, see "Media Types" (page 45), "File Systems" (page 36), and "Networking Extensions" (page 35). For detailed information on hardware support, see the installation guide that comes with Mac OS X.

USB

USB (Universal Serial Bus) is a high-speed plug-and-play interface between a computer and add-on devices such as audio players, joysticks, keyboards, telephones, scanners, and printers. It supports a data speed of 12 megabits per second. USB permits users to add a new device to their computer without having to add an adapter card or even having to turn the computer off. Mac OS X includes USB drivers for the following classes of devices:

- input devices (HID class)
- printers
- modems and other communication devices
- mass storage (Zip and Jaz drives, for instance, and external hard drives)
- imaging
- display
- audio

FireWire

FireWire is Apple's implementation of the IEEE 1394 standard (High Performance Serial Bus) for peripheral devices. It enables a single plug-and-socket serial connection on which up to 63 devices can be attached. Because it supports a data transfer rate up to 400 megabits per second, FireWire is ideal for devices such as digital cameras, DVDs, digital video tapes, digital camcorders, and music synthesizers. With FireWire, users can chain devices together in different ways without the need for terminators or complicated set-up requirements. And devices can be plugged in and used without the need for a system restart. Because IEEE 1394 is a peer-to-peer interface, you can connect one FireWire-capable device to another and use both without connecting either to a computer; for example, one camcorder can dub to another.

Velocity Engine

Support for the Velocity Engine is another important feature of Mac OS X. The Velocity Engine boosts the performance of any application exploiting data parallelism, such as those performing 3D graphic imaging, image processing, video processing, audio compression, and software-based cell telephony. Quartz, QuickTime, and QuickDraw now incorporate Velocity Engine capabilities; thus any application using these APIs can tap into the Velocity Engine without making any changes. The Mac OS X SDK includes a C/C++ compiler with Velocity Engine support so you can also create new applications that take full advantage of the Velocity Engine.

AirPort

AirPort is Apple's wireless network technology that delivers fast and reliable communications between multiple computers in a local area network and between that network and the Internet. With AirPort, several users can be online at the same time—simultaneously surfing the Web, accessing email, competing in games, and swapping files—all through a single Internet service account. AirPort also lets you wirelessly transfer files from your computer to another AirPort-equipped iBook, iMac, PowerBook, or Power Mac G4 from up to 150 feet away.

The wireless data rate for AirPort is 11 megabits per second for up to 10 simultaneous users per base station. Because it is based on the IEEE 802.11 Direct Sequence Spread Spectrum (DSSS) worldwide industry standard, AirPort permits interoperability with other 802.11-based equipment. And because AirPort uses radio signals, it can communicate through solid objects.

Video Features

The Quartz Compositor in Mac OS X includes a hardware acceleration layer called Quartz Extreme. This transparent layer supports the rendering of video, 2D, and 3D graphics using OpenGL-based video hardware. Support for this acceleration is automatic and does not require any special coding.

Mac OS X also supports the ability to "hot-swap" monitors, that is, to change monitors without shutting down the host computer first. This feature provides users with more flexibility in setting up and using their computer. However, applications that rely on the current video settings need to be aware that those settings can now change at runtime. The Core Graphics framework (`CoreGraphics.framework`) defines an API that allow applications to register for notifications when the video settings change.

For more information on graphics and rendering, see "The Graphics and Windowing Environment" (page 67).

System Architecture

A key consideration in the design of Mac OS X was the need to integrate a diverse collection of technologies—some with greatly different histories—and base this unified set of technologies on an advanced kernel environment. This chapter explores the general outlines of the architecture that made this possible.

The central characteristic of the Mac OS X architecture is the layering of system software, with one layer having dependencies on, and interfaces with, the layer beneath it (see Figure 3-1 (page 55)). Mac OS X has four distinct layers of system software (in order of dependency):

■ **Application environments**. Encompasses the five application (or execution) environments: Carbon, Cocoa, Java, Classic, and BSD Commands. For developers, the first three of these environments are the most significant. Mac OS X includes development tools and runtimes for these environments.

  See "Application Environments" (page 59) for more information.

■ **Application Services**. Incorporates the system services available to all application environments that have some impact on the graphical user interface. It includes Quartz, QuickDraw, and OpenGL as well as essential system managers.

  See "The Graphics and Windowing Environment" (page 67) and "Other Application Services" (page 78) for more information.

■ **Core Services**. Incorporates those system services that have no effect on the graphical user interface. It includes Core Foundation, Open Transport, and certain core portions of Carbon.

  See "Core Services" (page 80) for more information.

- **Kernel environment**. Provides the foundation layer of Mac OS X. Its primary components are Mach and BSD, but it also includes networking protocol stacks and services, file systems, and device drivers. The kernel environment offers facilities for developing device drivers (the I/O Kit) and loadable kernel extensions, including Network Kernel Extensions (NKEs).

  For further information, see the section "A Layered Perspective" (page 54) and the book *Inside Mac OS X: Kernel Programming*.

The Core Services and Application Services layers and the Carbon and Cocoa application environments are packaged in umbrella frameworks (described in the chapter "Umbrella Frameworks" (page 153)). Many public APIs of the kernel environment are exported through headers found in `/usr/include`.

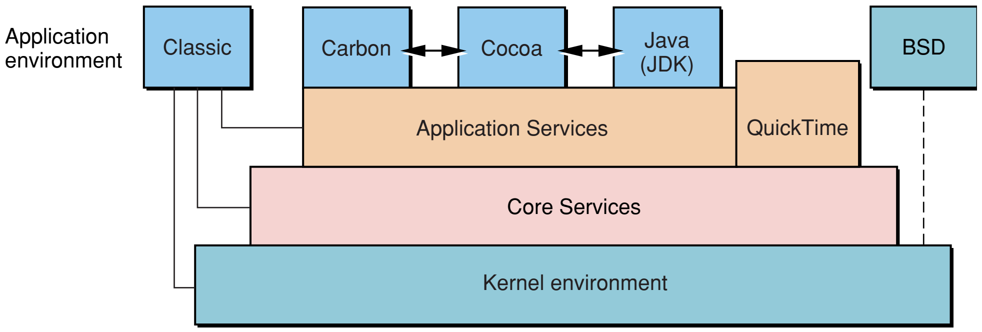
The first part of this chapter, as summarized in the foregoing paragraphs, presents the architecture of Mac OS X as layers of system software. Following this static perspective of Mac OS X is a more dynamic view that traces the progress of a user event through the system. A typical event in Mac OS X originates when the user manipulates an input device such as a mouse or a keyboard. The device driver associated with that device, through the I/O Kit, creates a low-level event, puts it in the window server's event queue, and notifies the window server. The window server dispatches the event to the appropriate run-loop port of the target process. There the event is picked up by the Carbon Event Manager and forwarded to the event-handling mechanism appropriate to the application environment. Events can also be asynchronous, such as a network packet containing configuration changes.

A Layered Perspective

A common way to look at complex software is to separate out parts of that software into "layers." Visually depicted, one layer sits on top of another, with the most fundamental layer on the bottom. This kind of diagram suggests the general interfaces and dependencies between the layers of software. The higher layers of software, which are the closest to actual application code, depend on the layer immediately under them, and that intermediate layer depends on an even lower layer.

Mac OS X is reducible to such a perspective. Figure 3-1 illustrates the general structure of Mac OS X system software as interdependent layers of libraries, frameworks, and services.

Figure 3-1 Mac OS X as layers of system software



Although this diagram does help clarify the overall architecture, there are dangers in the necessarily over-simplified view it presents. The Mac OS X services and subsystems that one application uses—and how it uses them—can be very different from those used by another application, even one of a similar type. Dependencies and interfaces at the different levels can vary from program to program depending on individual requirements and realities.

With that caveat aside, let's take a guided tour through the layers depicted in this diagram.

The boxes in the top row of the diagram of Figure 3-1 represent the different application (or execution) environments of Mac OS X. There are five such environments. The Classic and the BSD Commands environments are unique in the way they interact with the underlying layers of the system:

- The Classic "compatibility" environment is where users can run their Mac OS 8 or Mac OS 9 applications. Instead of sitting on top of the Application Services, the Classic environment in this diagram has lines connecting it to each layer. These connections indicate that the Classic environment is "hard-wired" into Mac OS X; it is not an environment for which developers can specifically compile code in Mac OS X. In other words, there are no public non-Carbon Mac OS 8 or Mac OS 9 APIs on a Mac OS X system that can be compiled. For further information on the Classic environment, see "The Classic Environment and Your Application" (page 239) in the chapter ""Installation and Integration" (page 223)".

- The BSD Commands environment provides a shell in which you can execute BSD programs on the command line. The standard BSD tools, utilities, and scripts are available for this environment as well as any custom ones you or third parties create. The diagram shows the BSD Commands environment connected directly with the kernel-environment layer. Note that you can run programs on the command line that are built in non-BSD environments, such as programs based on Cocoa's Foundation framework.

  The kernel environment exports BSD services to the upper layers of the system through the System library in `/usr/lib` (the headers are located in `/usr/include`). BSD commands are also available to developers; however, BSD commands might not be included in certain Mac OS X installations. Because the BSD Commands environment is a special optional environment, it is not described further in this document.

Carbon, Cocoa, and Java are the three principal application environments for Mac OS X developers:

- Carbon is an adaptation of the Mac OS 9 APIs and libraries for Mac OS X. It carries over most of the prior APIs (70 percent of the functions) and includes some APIs and services specifically developed for Mac OS X. See "Carbon" (page 60) for a discussion of Carbon.
- Cocoa is a collection of advanced object-oriented APIs for developing applications written in Java and Objective-C. See "Cocoa" (page 63) for more information on Cocoa.
- The Java environment is for the development and deployment of 100% Pure Java and mixed-API Java applications and applets. See "Java" (page 64) for an overview of this application environment.

Directly supporting the Carbon, Cocoa, and Java environments are the layers of system software that offer services for all application environments. These layers are stacked in decreasing widths to suggest that application code can access lower layers directly—that is, without the mediation of intervening layers. (However, see the warning about linking outside of umbrella frameworks in "Restrictions on Subframework Linking" (page 160) in the chapter ""Umbrella Frameworks" (page 153).")

The first of these layers is the Application Services layer. It contains the graphics and windowing environment of Mac OS X, principally implemented by Quartz and QuickDraw. This environment is responsible for screen rendering, printing, event handling, and low-level window and cursor management. It also holds libraries, frameworks, and background servers useful in the implementation of graphical user interfaces. See "The Graphics and Windowing Environment" (page 67) and "Other Application Services" (page 78) for details.

QuickTime is an extension to the operating system that architecturally spans layers of system software. It is an interactive multimedia environment that has features and functionality common to both a graphics environment and an application environment. Figure 3-1 (page 55) presents QuickTime as straddling the line between Application Services and the application environments. QuickTime requires a host application environment (or a browser) in which to execute, but the multimedia components that it offers have unique and sophisticated capabilities typically found only in application environments.

The Application Services layer sits on top of Core Services. In the Core Services layer are the common services that are not directly a part of a graphical user interface. Here you find cross-environment implementations of basic programmatic abstractions such as strings, run loops, and collections. There are also APIs in Core Services for managing processes, threads, resources, and virtual memory, and for interacting with the file system. "Core Services" (page 80) discusses this layer of system software.

The kernel environment is the lowest stratum of system software, just below the Core Services layer. The kernel environment provides essential operating-system functionality to the layers above it, such as

- preemptive multitasking
- advanced virtual memory with memory protection and dynamic memory allocation
- symmetric multiprocessing
- multi-user access
- file systems based on VFS (Virtual File System)
- device drivers
- networking
- basic threading packages

It is a high-performance and highly modular kernel with support for dynamic loading of device drivers, networking extensions, and file systems.

The kernel environment consists of five major components:

- **Mach**. Provides the fundamental abstractions and implementations of tasks, threads, ports, virtual addressing, memory management, and intertask communication. Mach is also the part of the operating system that manages processor usage, handles scheduling, and enforces memory protection. In addition it provides timing services, synchronization primitives, and a messaging-centered infrastructure to the rest of the operating system.
- **BSD**. A version of 4.4BSD is used, among other things, to support Mach's preemptive multitasking, memory protection, dynamic memory allocation, and symmetric multiprocessing. BSD forms the basis for networking and file systems in Mac OS X. Some of the other facilities it provides or supports are process creation and management, signals, system bootstrap and shutdown, generic I/O operations, basic file operations, and handling of terminals and other devices. It also implements user and group IDs as well as the related features of resource limits and access policies for files and other resources. BSD provides many of the POSIX APIs.
- **Device drivers and the I/O Kit**. Device drivers in Mac OS X are created with the I/O Kit, a framework that offers an object-oriented programming model (based on a restrictive form of C++) to streamline the development of device drivers. The I/O Kit takes into account underlying operating-system features such as virtual memory, memory protection, and preemption and thus relieves device-driver writers from having to worry about them in their code. It is designed to be modular, reusable, and extensible. The kernel environment includes a number of ready-made device drivers (see the chapter "System Technologies" (page 23)).
- **Networking**. The kernel environment implements numerous native networking protocols and facilities, which are described in "Networking and the Internet" (page 45) in the chapter ""System Technologies" (page 23)." Some of the networking facilities and protocol stacks of Mac OS X are implemented as Network Kernel Extensions (NKEs). They can extend the networking infrastructure of the kernel dynamically—that is, without recompiling and relinking the kernel.
- **File systems**. The kernel environment supports many different file systems and volume formats, including Mac OS Extended (HFS+), Mac OS Standard (HFS), UFS, NFS, and ISO 9660 for CD-ROMs. Mac OS Extended is the default file system, and Mac OS X typically boots and "roots" from it (that is, the kernel uses

the file system on an HFS+ volume as the one to mount first). By using the Virtual File System (VFS) infrastructure, developers can write kernel extensions that add support for other file systems. VFS is a set of standard, internal file-system interfaces and utilities for building such extensions. For summaries of the supported formats, see "File Systems" (page 36) in the chapter ""System Technologies" (page 23)."

As described in "Darwin and Open Source Development" (page 37) in the chapter ""System Technologies" (page 23)," the kernel environment is a subset of Darwin, Apple's Open Source technology. Darwin combines the Mac OS X kernel environment and the BSD commands and libraries essential to the BSD Commands environment. For more on the Mac OS X kernel environment and its relation to Darwin, see the document *Inside Mac OS X: Kernel Programming*.

The kernel environment, Core Services, and Application Services layers of Mac OS X are packaged as umbrella frameworks. Two of the primary Mac OS X application environments, Carbon and Cocoa, are also packaged as umbrella frameworks. See the chapter "Umbrella Frameworks" (page 153) for more about this subject.

Application Environments

An application environment consists of the frameworks, libraries, and services (along with associated APIs) necessary for the runtime execution of programs developed with those APIs. The application environments have dependencies on all underlying layers of system software.

Mac OS X currently has five application environments: Classic, BSD Commands, Carbon, Cocoa, and Java. This section provides overviews of Carbon, Cocoa, and Java.

Carbon

Carbon is a set of programming interfaces derived from earlier Mac OS APIs that have been modified to work with Mac OS X, especially its kernel environment. Carbon carries forward most of the existing Mac OS managers and APIs; specifically, this entails about 70 percent of the total functions and 95 percent of functions used by typical applications.

The Carbon APIs are too large and complex to summarize adequately here. However, some of the major differences between Carbon and its Mac OS predecessors are worth noting.

Memory. In adaptation to the kernel environment's features of advanced virtual memory and memory protection, many APIs—particularly the Memory Manager—have undergone changes that restrict or eliminate the use of zones, system memory, or temporary memory. For example, temporary memory allocations in Mac OS X are allocated in the application's address space. Although there are no longer functions for accessing the system heap, new routines are provided for the allocation of shared and persistent memory. In addition, the virtual memory system in Mac OS X introduces a number of changes in the addressing model. (See *Inside Mac OS X: Kernel Programming* for information on this subject.)

Hardware Interfaces. The Mac OS 9 managers used for low-level access to hardware—for example, the ADB Manager, the Device Manager, and the Ethernet Driver—are not implemented in Mac OS X. The different device-driver architecture provided by the I/O Kit mediates all low-level access to hardware devices.

Resources. Because there is no ROM in Mac OS X, functions related to accessing resources in ROM are unsupported in Carbon. Also the Resource Manager places greater restrictions on accessing the resource map.

Required Replacement Managers. The Carbon technologies listed in Table 3-1 now take the place of their predecessors. Use of the new libraries is required.

Table 3-1 Required Manager replacements for Carbon

Instead of	Now use
AppleTalk Manager	Open Transport
PPC Toolbox	Apple events
Standard File Package	Navigation Services
QuickDraw 3D	OpenGL
Help Manager	Carbon Help Manager
Apple Guide	Apple Help
Printing Manager	Carbon Printing Manager
Any device manager	I/O Kit
QuickDraw GX	Quartz and Apple Type Services for Unicode Imaging
Vertical Retrace Manager	Time Manager

Recommended Replacement Managers. The Carbon technologies listed in Table 3-2 provide more robust support than their predecessors. Use of the new Carbon libraries is preferred, but not required.

Table 3-2 Recommended Manager replacements for Carbon

Instead of	Now use
TextEdit	Multilingual Text Engine
QuickDraw	Quartz
Event Manager	Carbon Event Manager
Font Manager	Apple Type Services for Fonts

Table 3-2 Recommended Manager replacements for Carbon (continued)

Instead of	Now use
QuickDraw Text	Multilingual Text Engine or Apple Type Services for Unicode Imaging
Resource Manager	Interface Builder Services
Script Manager	Unicode Utilities

General Changes. Many functions in the various managers have been changed or removed throughout Carbon. (See the *Carbon Specification* for complete details.)

- **Data structures**. To ensure the integrity of system data and to support access to all system services through preemptive threads, Carbon restricts direct access to data structures. Instead of functions that return pointers or handles to structures that can be dereferenced, Carbon now supplies accessor functions for getting and setting field data. In addition, it includes functions for creating and disposing of data structures.
- **Definition procedures**. The Window Manager, Menu Manager, Control Manager, and List Manager in Carbon still permit you to create and use standard and custom definition procedures (WDEFs, MDEFs, CDEFs, and LDEFs), but you must be sure to compile them as PowerPC code. Additionally, these managers provide new routines for creating and packaging them.
- **68K code**. Mac OS X does not support 68K code (except in the Classic environment). For this reason the Trap Manager (and the trap table), the Mixed Mode Manager, and the Patch Manager are unavailable or greatly reduced in scope in Carbon. For the same reason, many other functions have been dropped from Carbon.

Many of the commonly used parts of Mac OS X are Carbon managers or are daemons, applications, or frameworks created with Carbon APIs. For example, the system processes that handle events and manage application processes in Mac OS X are Carbon managers, many of the managers in the Core Services layer are Carbon-based (see "Core Services" (page 80)), and the Finder is a Carbon application.

For more information on Carbon, consult the Carbon documentation website at http://developer.apple.com/techpubs/Carbon/carbon.html. In particular, see the documents *Inside Carbon: Carbon Porting Guide*, which contains specific information about converting code to the Mac OS X application model, and the *Carbon Specification*, which gives details on which managers and functions are supported in Carbon.

Cocoa

The Cocoa application environment is based on two object-oriented frameworks: Foundation (`Foundation.framework`) and the Application Kit (`AppKit.framework`). These frameworks offer both Java and Objective-C APIs (with most Java classes simply "bridging" to their Objective-C implementation).

Foundation and the Application Kit are similar in some respects to the Core Services and Application Services layers, respectively. The classes in the Foundation framework provide objects and functionality that have no impact on the user interface; Foundation is directly based on Core Foundation. The classes of the Application Kit furnish all the objects and behavior that affect what users see in the user interface, such as windows and buttons, and responsiveness to their mouse clicks and key presses. The Application Kit directly depends on Foundation.

The Foundation framework's classes fall into several categories:

- object wrappers (or "helpers") for basic programmatic types and operations, including strings, arrays, dictionaries, numbers, byte swapping, parsing, and exception handling
- object wrappers for kernel-environment entities and services, such as tasks, ports, run loops, timers, threads, and locks
- object-related functionality, particularly memory management (autorelease pools), remote invocations, archiving, and serialization
- file-system and I/O functionality including URL handling, file seeking, and dynamic loading of code and localized resources
- other services, such as distributed notifications, undo (and redo), data formatting, and dates and times

Many of the Application Kit's classes, as might be expected, are designed for the creation and management of objects that appear in a graphical user interface. Among these are classes for windows, dialogs, buttons, tables, text fields, sliders, pop-up menus, scroll views, application (pull-down) menus, and even a movie view for QuickTime streaming.

However, the Application Kit has features and functionality that make it far more useful than just a collection of classes for user-interface objects.

- It has sophisticated mechanisms for event handling and application and document management.
- It gives applications ways to integrate and manage colors, fonts, and printing (even providing the dialogs for these features).
- It allows you to composite images in many different graphical formats and it offers a framework for drawing, including the application of vector transformations.
- It includes facilities for spell checking, dragging, and copy-and-paste operations.

Other Cocoa frameworks are also available for scripting, network management, and other purposes.

The Cocoa umbrella framework (`Cocoa.framework`) imports both Foundation and the Application Kit. If you are writing an application, link with the Cocoa framework. If you are writing any Cocoa program that does not have a graphical user interface (a background server, for example), you should link at least with the Foundation framework.

For more information about Cocoa, see the Cocoa documentation on the Apple website: http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html.

Java

The Java application environment allows you to develop and execute Java programs on Mac OS X, including 100% Pure Java applications and applets. This environment is implemented in conformance with an industry standard—that is, a recent version of the Java Development Kit (JDK) including the Java virtual machine (VM). Because of this, a Java application created with this environment is very portable. You can copy it to a computer that has entirely different hardware and a different operating system and, as long as that system includes a compatible version of the Java VM, your application should run on it. A Java applet should run in any Internet browser with the proper capabilities.

Note: The Cocoa application environment includes Java packages corresponding to the Application Kit and Foundation frameworks. These packages allow you to develop a Cocoa application using Java as the development language. You can mix (within reason) the APIs from these packages and native Java APIs (excluding AWT or Swing APIs). For more on the Cocoa application environment, see "Cocoa" (page 63). In addition, Apple's JDirect and Sun's JNI (Java Native Interface) programming interfaces allow your Java programs to call other frameworks, including Carbon. And you can write multimedia Java applications for the Mac OS and Windows platforms using QuickTime for Java.

The Java application environment in Mac OS X has three major components:

- A development environment, including the Java compiler (`javac`) and debugger (`jdb`) as well as other tools, including `javap`, `javadoc`, and `appletviewer`.

  This "command-line" environment requires a BSD shell, such as that provided by Apple's Terminal application. Apple supplies the Project Builder application as a front-end to this environment and third parties may supply their own front-ends. The command-line tools are located in the `JavaVM.framework/Commands` subdirectory, with symbolic links supplied to this directory in `/usr/bin`.

- A runtime environment consisting of Sun's high-performance Hotspot Java virtual machine, the "just-in-time" (JIT) bytecode compiler, and the basic Java packages.

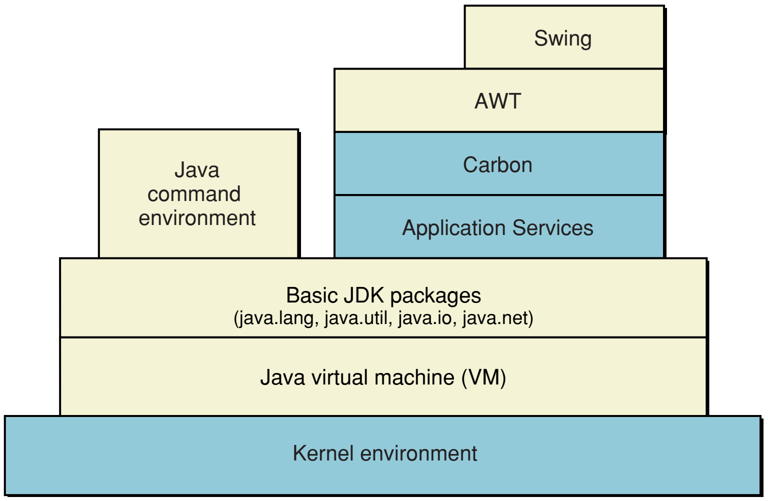
  The Java virtual machine is located at `/System/Library/Frameworks/JavaVM.framework/Libraries`. The basic packages include `java.lang`, `java.util`, `java.io`, and `java.net`; they are in the `classes.jar` archive in the `Classes` directory of the same framework.

- An application framework containing the classes necessary for building a Java application.

  The more significant of these packages are `java.awt` and `javax.swing`, commonly known as AWT (Abstract Windowing Toolkit) and Swing. The AWT package implements standard user-interface components (such as buttons and text fields), basic drawing capabilities, a layout manager, and the event-handling mechanism. The Swing package provides a greatly extended set of user interface components. These components automatically take on the look and feel of the host platform. Swing includes versions of the existing AWT component set plus a rich set of higher-level components, such as tree view, list box, and tabbed panes.The AWT and Swing package are in a jar archive located at `JavaVM.framework/Classes/classes.jar`.

Figure 3-2 shows a more detailed view of the Java environment.

Figure 3-2 Architecture of the Java environment


Swing
AWT
Carbon
Application Services
Java command environment
Basic JDK packages (java.lang, java.util, java.io, java.net)
Java virtual machine (VM)
Kernel environment

The Java virtual machine along with the basic Java packages—`java.lang`, `java.util`, and `java.io`—are equivalent to the Core Services layer of system software for the Carbon and Cocoa environments. They draw on the resources of the kernel environment to implement low-level services such as process management, threading, and input/output. They do not need to access anything in the Core Services layer of system software (Open Transport, Core Foundation, and so on).

All other parts of Java in Mac OS X are layered on top of the VM and the basic packages. If a Java program does not have a user interface (say, a tool or an application server), all it needs is this foundation to execute. But a 100% Pure Java application or applet (which, by definition, has a graphical user interface) must use AWT or Swing, both of which bind with many of the frameworks and libraries in the Application Services layer of system software. Swing itself is layered on a primitive part of the AWT package. AWT and Swing together are architecturally equivalent to a GUI-oriented toolbox or framework such as Carbon's Human Interface Toolbox or Cocoa's Application Kit.

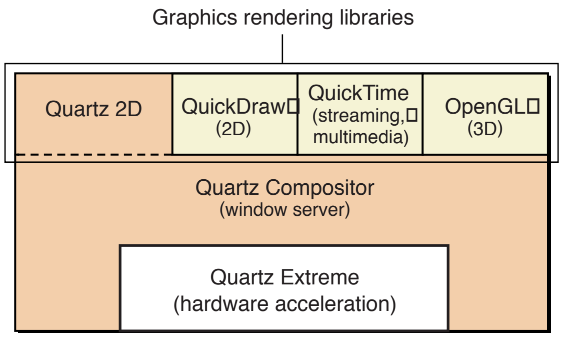
Java applications can be double-clickable bundles just as any Carbon or Cocoa application is. (You use the MRJAppBuilder utility to bundle Java applications.) They can also be executables that users must run from the command line or that are executed through the system `exec` call or the Java `Runtime.exec` method. In the latter case, the Java tool used to launch the executable (`java` or, for applets, `appletviewer`) is displayed as the process name (for example, in the Process Viewer).

The Graphics and Windowing Environment

The preeminent application services of Mac OS X are those that make up the graphics and windowing environment. An application, by its very nature, must display its windows in a graphical user interface and allow users to manipulate its controls. A graphics and windowing environment confers these basic capabilities on applications "for free," relieving them of the burden of implementing them on their own. In addition to rendering text and images in windows on a screen (as well as printing them), this environment also provides essential low-level facilities such as initial event routing and cursor management.

The core portion of the Mac OS X graphics and windowing environment is called Quartz. As depicted in Figure 3-3, Quartz has two parts: Quartz 2D and Quartz Compositor. (The Quartz Extreme layer is integrated into Quartz Compositor.)

Figure 3-3 Quartz and the graphics and windowing environment


Graphics rendering libraries
Quartz 2D
QuickDraw (2D)
QuickTime (streaming, multimedia)
OpenGL (3D)
Quartz Compositor (window server)
Quartz Extreme (hardware acceleration)

The Quartz 2D part of Quartz is one of several graphics libraries that provide graphics-rendering services. It is designed for the display of two-dimensional text and graphics. Peer graphics and multimedia libraries include

- QuickDraw for rendering two-dimensional images
- OpenGL for rendering both two- and three-dimensional images
- QuickTime for rendering streaming digital video and other multimedia

  QuickTime is an interactive multimedia environment that includes capabilities and features found in both a graphics environment and an application environment. Despite its hybrid status in the Mac OS X architecture, this section, as a simplification, treats it as a peer graphics library to Quartz 2D, QuickDraw, and OpenGL.

All of the rendering libraries have direct dependencies on the other part of Quartz, the Quartz Compositor layer. However, QuickTime and OpenGL have fewer dependencies because they implement their own versions of certain windowing capabilities.

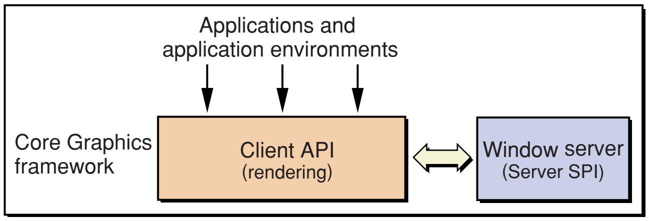
Quartz Compositor consists of the Mac OS X window server and the (currently private) system programming interfaces (SPIs) it implements. The window server has overall responsibility for displays and windows, including their composition, positioning, and basic management. It also performs low-level cursor management and event routing.

Quartz is largely implemented in the Core Graphics framework (`CoreGraphics.framework`). The dynamic shared library of this framework, as illustrated by Figure 3-4, includes the client APIs; the server SPIs are implemented by the window server itself. Applications or application environments link with the client side of the library—Quartz 2D—for screen rendering, PDF generation, and other services. All access to the server SPIs is mediated through the client APIs.

The Cocoa and Java environments provide their own programming interfaces to Quartz 2D and, to some extent, the other rendering libraries. You may use either the Cocoa and Java interfaces, or you may use the programming interfaces in the Application Services layer.

The remainder of this section discusses the role of Quartz in the graphics and windowing environment. For conceptual information on QuickDraw, QuickTime, and OpenGL, consult the relevant Apple developer documentation (`developer.apple.com`).

Figure 3-4 The Core Graphics framework


Applications and application environments
Core Graphics framework
Client API (rendering)
Window server (Server SPI)

Quartz Compositor

The Quartz Compositor layer of Mac OS X comprises the window server and the (private) system programming interfaces (SPI) implemented by the window server. In this layer are the facilities responsible for rudimentary screen displays, window compositing and management, event routing, and cursor management.

The window server is a single system-wide process that coordinates low-level windowing behavior and enforces a fundamental uniformity in what appears on the screen. It is a lightweight server in that it does not do any rendering itself, but instead communicates with the client graphics libraries layered on top of it. It is "agnostic" in terms of a drawing model.

The window server has few dependencies on other system services and libraries. It relies on the kernel environment's I/O Kit (specifically, device drivers built with the I/O Kit) in order to communicate with the frame buffer, the input infrastructure, and input and output devices. It also links with certain frameworks in Core Services to acquire process-management services such as basic process activation.

One of the primary duties of the window server is window compositing. It composites and recomposites each pixel of an application's window as the window is drawn, redrawn, covered, and uncovered. Each window is represented as a bitmap that includes both translucency (alpha channel) and anti-aliasing information. The bitmap serves as a buffer, allowing the window server to "remember" an application's window contents and to recomposite it without the application's involvement. However, the window server (Quartz Compositor) does not retain vector information that a graphics library (such as its own Quartz 2D) might have used to create a window or any other image. Instead, the bitmap is shared between the application and window server; the application draws directly into the shared bitmap and the window server composites the bitmap appropriately.

In its Quartz Compositor component, Quartz models the windowing system as a *layered* compositing engine. Traditional windowing systems use a "switch" model in which every pixel on a screen belongs entirely to one window (or the desktop). Because of this model, transitions are necessarily abrupt; when you close a window, for example, it disappears immediately. A layered compositing window system, on the other hand, is based on a "video mixer" model in which every pixel on the screen—particularly in the attributes of translucency and anti-aliasing—can be shared among windows in real time. This model allows for smooth transitions between the states of a graphical user interface, one of the distinctive characteristics of the Aqua experience.

For the role of the window server in event handling, see "Tracking a User Event" (page 86).

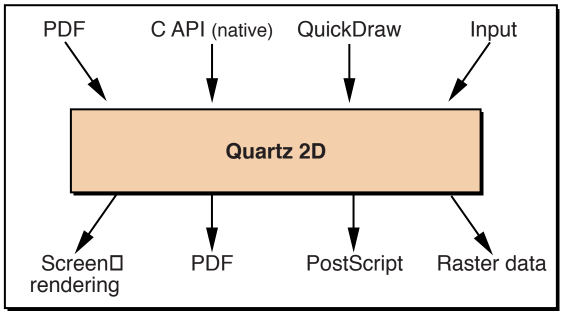
Quartz 2D

The Quartz 2D part of Quartz is a graphics library with a vector flavor. Its APIs allow you to create text and images by specifying a sequence of commands and mathematical statements that place lines, shapes, color, shading, translucency, and other graphical attributes in two-dimensional space. You do not need to specify the attributes of individual pixels. As a result, a shape can be efficiently defined as a series of paths and attributes rather than as a bitmap.

By using vectors, Quartz 2D can also use a coordinate system for drawing based on, say, inches or centimeters rather than a pixel grid. The coordinate system is flexible, permitting various measurement standards, and it enables some degree of display independence since it is not bound to any screen resolution. It also uses floating-point coordinates. Prior to compositing by Quartz Compositor, Quartz 2D translates the vector information of an image, which is described in terms of the coordinate system, to pixel values.

The Quartz 2D API is device independent—that is, the final destination of the drawing operation may be a window's bitmap, but it may also be a Portable Document Format (PDF) file, a PostScript file, or another output format. An application may call the API directly or it may call it indirectly when displaying a PDF file, when using QuickDraw, or when using other input mechanisms. Figure 3-5 illustrates this relationship.

Figure 3-5 Quartz 2D inputs and outputs



The primary inputs for Quartz 2D are the drawing commands and statements made with QuickDraw and the native C APIs. (Future APIs in the front-end may be supported.) Applications using QuickDraw can call into Quartz 2D through a `CGContextRef` interface and thereby get its capabilities. QuickDraw enables them to obtain the `CGContextRef` from a `GrafPort` interface.

The commands and statements from QuickDraw or the native APIs are immediately converted to the required output format, whether that be bitmap data for screen rendering, PostScript (for PostScript printers), or raster data for other types of printers. The PDF can also be published "as is"; this happens automatically for print preview. Future back-end converters, such as for plotters, may be supported.

Quartz 2D, as the foregoing paragraph suggests, is the underlying engine for the Mac OS X printing system. Printing is often a two-pass affair. Quartz 2D interprets the text and images constructed with the native C or QuickDraw APIs and stores them in PDF form (the primary spooling format). Then this PDF is fed through Quartz 2D again to convert it to the appropriate output format.

The Printing System

The Mac OS X printing system provides a flexible and powerful new printing environment for Macintosh developers. The architecture makes it much easier for application developers to support printing in their applications and for printer vendors to write printer drivers and extend printing dialogs. The Mac OS X printing system has a number of advantages over the printing system used in Mac OS 8 and 9, including the following:

- The printing system uses Quartz 2D for rendering and conversion services. Quartz 2D supports a resolution-independent PDF drawing model that allows applications to print high-quality, color-managed output on all classes of raster and PostScript printers.
- Sheets allow you to open more than one printing dialog at the same time and to send more than one print job to a printer queue.
- Printing dialogs can be customized with printing dialog extensions. This means you do not need to completely replace the standard printing dialogs to provide users with application-specific or printer-specific features. The Page Setup dialog can be extended by application developers, while both printer vendors and application developers can extend the Print dialog.
- Printer modules replace printer drivers. Printer modules are easier to write than printer drives, as much of the code that has to be in a driver is now taken care of for you in Apple-supplied I/O modules and other parts of the printing system.

- The printing application programming interface (API) includes robust support for Carbon applications. Carbon developers can write one application that can run in Mac OS X as well as in Mac OS 8 and 9. Cocoa developers support printing by using Cocoa objects and methods. Cocoa methods call through to the Carbon Printing Manager API.

A key aspect of the new printing system's design is its robust support for Carbon applications. Because the Carbon Printing Manager is supported in Mac OS 8 and 9 as well as Mac OS X, a Carbon application is able to print as expected in both environments. For example, when running Mac OS 8 or Mac OS 9, the application utilizes the traditional user interface and drivers. On Mac OS X, the application automatically takes advantage of the new printing system's more consistent set of printing dialogs and flexible printing architecture.

For more information about printing, see *Inside Mac OS X: About the Mac OS X Printing System*.

The User Interface of the Printing System

The Mac OS X printing system's user interface provides a consistent, easy-to-use environment for performing printing-related tasks such as locating local and networked printers, configuring new printers, choosing printers, and managing print jobs. The new printing system's human interface allows users to handle simple, everyday printing tasks and complex, multidocument, multiprinter print jobs.

The printing system's user interface consists of the following components:

- **Print Center**. Allows the user to locate, select, and configure available printers, and to determine the status of print jobs associated with each.
- **Page Setup dialog**. Allows the user to specify the format of the document to be printed.
- **Print dialog**. Allows the user to specify the parameters of a print job, and to print a document on a specified printer.
- **Simplified print dialog**. Allows the user to specify common print job parameters without navigating to separate panes of the print dialog.

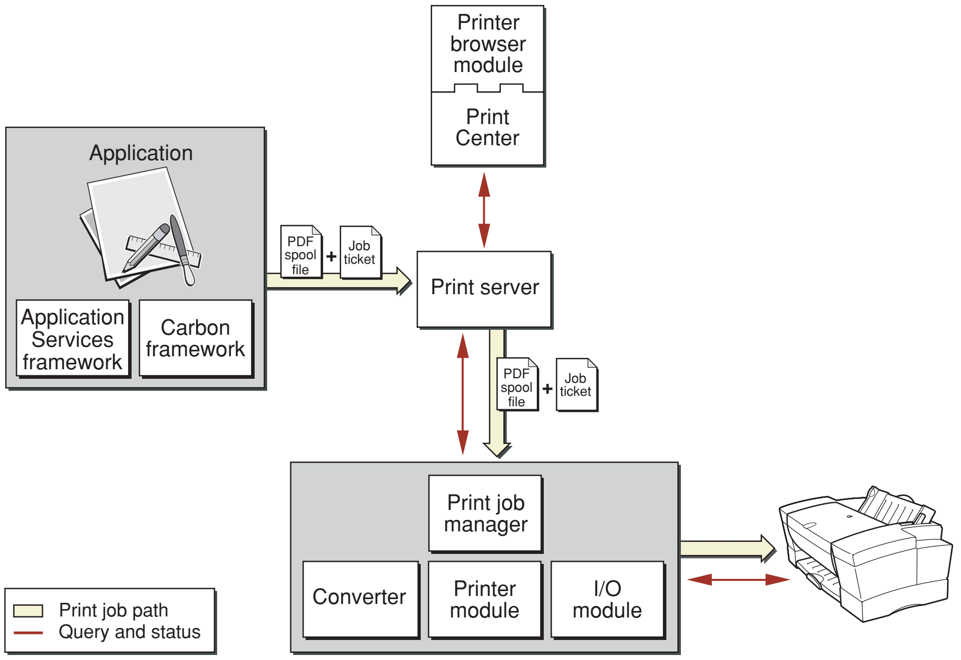
The new printing system's interface includes a number of important improvements in both ease-of-use and stability relative to the Mac OS 8 and 9 printing model. The Chooser—the most common source of user confusion when dealing with printers—

is replaced by Print Center, which combines many of the features of the Chooser and desktop printing into a single, integrated interface. Unlike the Chooser, Print Center is a separate application from the Finder, which eliminates the need for the Finder to support the printing interface, simplifying code and improving system stability. The Page Setup and Print dialogs are standardized for all printers and are easily extensible to allow for third-party customization.

Summary of Printing Architecture

The Mac OS X printing architecture consists of a set of nine modules. Conceptually, these modules can be divided into client and server groupings. The Application Services framework and the Carbon framework are client modules that provide the drawing and printing APIs for applications. The Print Center and optional printer browser modules handle client interactions during the printing cycle. The remaining five modules—print server, print job manager, converters, printer modules, and I/O modules—constitute the printing system's server back end, which accepts print jobs from local clients and renders them to the destination printer. The printing modules and their relationships are depicted in Figure 3-6.

Figure 3-6 Mac OS X printing system



Here's a brief description of the modules shown in Figure 3-6.

- **Carbon framework** Implements the Carbon Printing Manager API used by applications. Displays the Print and Page Setup dialogs, captures drawing information from applications, and passes the data to the Print Server for printing.
- **Application Services framework** Contains common printing functions used by Carbon, Cocoa, and BSD UNIX applications. This framework does not implement any of the printing dialog interfaces, which are instead implemented in the Carbon framework.
- **Printing dialog extensions (PDEs)**. (Not shown on diagram.) Extends a Print or Page Setup dialog, allowing third parties to add user interface elements in support of specific printers. A PDE is paired with a printer module or Carbon application, which interprets and applies the custom settings offered by the PDE.

- **Print Center**. Allows the user to locate and select printers, as well as control and obtain status for print jobs.
- **Printer browser module (PBM)**. Extends Print Center by adding UI support for additional printer connection methods such as SCSI and FireWire. A PBM is paired with an I/O module, which implements support for the transport type.
- **Print server**. Handles the queuing of print jobs after they leave the application. Responds to requests from Print Center to manipulate or return status information about print jobs in the queue. Reports errors back to Print Center.
- **Print job manager**. Manages the various processes necessary to convert a single print job into final printed output. Hosts printer modules and I/O modules.
- **Converter**. Optional modules that assist the print job manager by transforming a print job's data format. A converter might transform PDF to raster, for example.
- **Printer module**. Formats data for the printer (PostScript or PCL, for example) and handles printer status and error conditions. Printer modules are typically created by printer vendors to support a particular printer or printer family.
- **I/O module**. Implements a standard interface for a transport type. Apple supplies modules for NetInfo, USB, TCP/IP, and AppleTalk. Third parties can also create modules to support additional transport types.
- **Job ticket**. Contains all the necessary parameters to control the printing of the job. Printing functions in the Application Services framework handle the creation of job tickets. Individual components update the job ticket status at each step in the printing process.

Printer Discovery

Before a user can choose a printer, Print Center must first compile a list of available printers. The process by which Print Center locates available printers is called "printer discovery."

During printer discovery, Print Center enumerates all of the I/O and printer browser modules installed in `/System/Library/Printers` and `/Library/Printers`. The print server retrieves string representations of the various connection types from the printer browser modules and passes them to Print Center. Print Center populates the connection pop-up menu with these strings.

When the user selects a connection type, Print Center enumerates all the printer modules installed in `/System/Library/Printers` and `/Library/Printers` and asks each printer module if it supports the chosen connection type. If so, Print Center retrieves icon and lookup information from it. The printer browser modules use this information as search criteria when searching for printers that support a connection type and as user interface elements in their display.

When the user clicks a printer to add it to the list of printers, Print Center gets the selected printer address, icon, and printer model information from the printer browser module. Print Center then uses this information to create a new print queue and add the printer to the list.

The Printing Process

An application stores information about a print job in one or more printer objects. Printer objects hold information about the print job, such as paper size and orientation, and are used to pass information to the print server. The printing system provides access to a default set of printer objects, which the application can modify. The Page Setup and Print dialogs use the printer objects to capture the user selections for the current print job.

At printing time, the application displays the Page Setup and Print dialogs in response to user requests. The application displays both of these dialogs using functions from the Carbon framework. Carbon application can extend a dialog using a printer dialog extension (PDE). Printer dialog extensions let the Carbon application define options specific to the application's drawing environment—custom page layouts, for example.

When the user dismisses the Print dialog, the Application Services framework accepts drawing commands (QuickDraw, Core Graphics, or a PDF file) from the application and puts them into a spool file. The application passes the spool file, along with any printer objects, to the printing system. Upon receiving the application data, the printing system creates a job ticket to manage the print-job settings and status and then passes the job ticket along with the spool file to the print server. The print server passes the data to the print job manager, which is responsible for managing the rest of the printing process. Upon sending the job to the print server, the application is done with its portion of printing. All errors related to the print job are now routed asynchronously back to the Print Center for display to the user.

The print job manager first consults the job ticket to determine the destination printer and queries the destination printer's associated printer module to find out what data format it requires. If necessary, the print job manager uses a converter to transform the incoming data into a format that the destination printer module can accept. Next, the print job manager passes the data to the printer module, which is responsible for converting the incoming data into the raw commands the printer will use to render the data. Finally the print job manager receives the printer-specific data from the printer module and uses the I/O module appropriate to the printer's connection type to send the data to the printer.

Other Application Services

The other system services in the Application Services layer support all application environments by supplying objects and behavior that affect the graphical user interface. This section discusses some of the more prominent of these services. Because of the evolving nature of Mac OS X, the composition of the Application Services layer will change over time. Check the subframeworks of the Application Services umbrella framework (`ApplicationServices.framework`) to learn what is currently included.

Process Manager

The Process Manager manages all processes in Mac OS X. It controls access to shared resources and manages the scheduling and execution of applications, allowing multiple applications to share CPU time and other resources. The Finder uses the Process Manager to launch applications when the user double-clicks an application or a document icon. The Process Manager also provides a number of routines that allow you to control the execution of processes, to launch processes, and to get information about processes.

For related information on the Process Manager, see "Tasks and Processes" (page 257) in the chapter ""Issues and Options With Multiple Environments" (page 257)."

Carbon Event Manager

The Carbon Event Manager dispatches events to the appropriate event-handler for that event, based on the type of event and the destination application environment. The window server puts an event it receives on the run-loop port of the target application process. The Carbon Event Manager gets the event from the port, packages it in an appropriate form, and gives it to the deepest "container" possible within the event-handling structure specific to the application (that is, Carbon, Cocoa, or Java). To do this, the Carbon Event Manager must often determine which window is currently the active one, whether there is keyboard focus in the window, and so on.

For more on event handling in Mac OS X, see "Tracking a User Event" (page 86).

Apple Events

An Apple event is a high-level event that applications can send to other applications on the same computer, on a remote computer, or even to themselves. Apple events are the primary mechanism for interapplication communication in Mac OS X. Applications typically use them to request services and information from other applications, or to provide services and information in response to such requests.

A related technology, the system-level scripting language AppleScript, is also part of Mac OS X. Users can use AppleScript to send Apple events to applications.

See "Interprocess Communication" (page 261) in the chapter ""Issues and Options With Multiple Environments" (page 257)" for further discussion of Apple events.

The Clipboard

The Mac OS X Clipboard (also known as the "pasteboard") is a background server that allows you to transfer data between applications. It is similar in some respects to the Mac OS 9 Clipboard, but different in others. The Mac OS X Clipboard can hold multiple representations of the same data. It is shared by all executing applications and contains data that the user has cut or copied, as well as other data that one application wants to transfer to another. It is used in copy-cut-paste operations and as the data-transfer mechanism in drag-and-drop operations. It is also used by Services to transfer data between applications.

Core Services

The Core Services layer contains the system services that do not have any effect on an application's graphical user interface. This level includes Core Foundation, Carbon Core, CFNetwork, web services, and Open Transport. The Core Services layer comprised primarily of two frameworks—the Core Services umbrella framework (`CoreServices.framework`) and the Core Foundation framework (`CoreFoundation.framework`). This section talks about the more prominent technologies in these frameworks; others (for example, core security services) are not discussed.

Core Foundation

Core Foundation is a framework (`CoreFoundation.framework`) that provides fundamental software services useful to application services, the application environments, and to applications themselves. Among the benefits of using Core Foundation is the increased capability for sharing code and data among frameworks, libraries, and applications in different environments and layers. Core Foundation also enables easy internationalization through Unicode strings and provides abstractions that contribute to operating-system independence.

Core Foundation uses the paradigm of opaque types; using these types, you can create "objects," each with its own individual identity and value (or set of values). It offers special facilities for allocating memory when these objects are created, and it has generic base types and polymorphic functions to facilitate intertype operations.

Core Foundation includes opaque types corresponding to such programmatic entities as strings, arrays, dictionaries, dates, numbers, and trees. It also features an architecture (and corresponding APIs) for plug-ins as well as a mechanism (with corresponding APIs) for dynamically finding and loading code and locale-dependent resources. Additionally, it has services for accessing local and remote resources via URLs, for setting up distributed notification centers, for reading and writing XML property lists, for parsing XML, and for writing and retrieving per-user and per-machine preferences.

Table 3-3 lists the Core Foundation services and their associated opaque types.

Table 3-3 Core Foundation services

Services	Types	Description
Base Services	CFAllocator, base types	Defines the base types and polymorphic functions that are used throughout the Core Foundation API.
Bundle Services	CFBundle	Offers an elegant means of organizing and locating many types of program resources including images, sounds, localized strings, and executable code.
Collection Services	CFArray, CFDictionary, CFTree, CFSet, CFBag	Provides high-level abstractions of common data structures—including arrays, dictionaries (associative arrays or vectors), and trees—along with associated functionality.
Notification Services	CFNotificationCenter	Implements distributed notifications, a mechanism that allows a process to send messages (notifications) to other processes on the same machine.
Plug-in Services	CFPlugIn	Provides a standard plug-in architecture for Mac OS X applications (as well as Mac OS 9 applications).
Preferences Services	CFPreference	Enables programs to store and retrieve user preferences. See "The Preferences System" (page 203) in the chapter ""Software Configuration" (page 197)" for background information.
Property List Services		Offers a way to organize data into a form that is meaningfully structured, transportable, storable, and accessible, but still as efficient as possible. The property list API allows you to convert hierarchically structured combinations of basic data types to and from standard XML.

Table 3-3 Core Foundation services (continued)

Services	Types	Description
Run Loop Services	CFSocket, CFRunLoop (and related)	Provides low-level event-handling and dispatch services.
String Services	CFString, CFCharacter-Set	Provides a full suite of fast and efficient string manipulation and conversion functionality. String Services offers seamless Unicode support and thus greatly simplifies internationalization. String Services also facilitates the sharing of string data between Carbon and Cocoa applications.
URL Services	CFURL CFURLAccess	Gives programs a way to access, via URLs, resources stored locally or remotely.
Utility Services	CFDate, CFTimeZone, CFNumber, CFUUID, CFByteOrder	Provides miscellaneous services such as date and time computation and representation, “object” wrapping of numbers, byte swapping, and UUID generation.
XML Parser	CFXMLParser	Provides a nonvalidating XML parser for reading and extracting data from XML documents.

Carbon Core

The Carbon Core is a part of `CoreServices.framework` that includes a number of Carbon managers and offers low-level services to all application environments. These services include cooperative and preemptive threading, resource management, memory management, and file-system operations. Table 3-4 summarizes these managers.

Table 3-4 Carbon managers in the Core Services layer

Manager	Description
Alias Manager	Helps locate specified files, directories, or volumes using aliases. It provides routines for creating and resolving file-system alias records.
Collection Manager	Provides an abstract data type for storing collections of information.
Component Manager	Enables your application to find and use various software objects (components) at runtime. Also allows your application to create and manage components.
Date, Time, and Measurement Utilities	Allows applications to obtain and manipulate information on dates, times, geographic location, time zone, and units of measurement.
File Manager	Gives programs the ability to access files stored on physical volumes, including hard disks, CDs, and Zip disks. It handles Mac OS Extended (HFS+), Mac OS Standard (HFS), UFS, NFS, and other supported file formats. The File Manager routines create, open, update, save, and close files; search for specific files or directories; obtain information about files or directories; and perform other advanced file-related operations. The File Manager supports Unicode and its APIs are thread-safe.
Folder Manager	Allows programs to find and search folders, create new ones, and control how files are routed between folders. It includes new support for domains.
Memory Management Utilities	Provides specialized routines useful for examining or controlling certain aspects of the memory environment.

Table 3-4 Carbon managers in the Core Services layer (continued)

Manager	Description
Memory Manager	Controls the dynamic allocation of memory within an application's protected address space. It includes new routines for allocating shared and persistent memory as well as functions related to the virtual memory system in Mac OS X.
Multiprocessing Services	Enables programs to create and manage separate preemptively scheduled threads. It also includes synchronization services and atomic instructions.
Resource Manager	Provides routines for creating, deleting, opening, reading, modifying, writing, and getting information about resource files. It includes support for data-fork based resources.
Text Encoding Conversion Manager	Provides two facilities—the Text Encoding Converter and the Unicode Converter—that applications can use to perform text conversions.
Text Utilities	Offers an integrated set of routines for performing a variety of operations on text, ranging from sorting strings to finding word boundaries.
Thread Manager	Enables programs to create and manage cooperatively scheduled threads.
Time Manager	Gives programs a way to schedule the execution of routines at a specified time, either once or repetitively. This mechanism for performing time-related tasks is hardware-independent.
Unicode Utilities	Performs various operations on Unicode text, including Unicode key translation.

CFNetwork

CFNetwork is a part of `CoreServices.framework` and is the preferred API to use for user-level networking and communications. CFNetwork provides a suite of functions for creating, serializing, deserializing, and managing protocol messages that are commonly exchanged between clients and servers. CFNetwork makes it unnecessary for you to have to learn and implement the details of a protocol in order to exchange protocol messages.

The design of CFNetwork supports a wide range of protocols, including HTTP, SMTP, LDAP, and FTP. The current implementation of CFNetwork provides functions for working with HTTP messages.

Web Services

Web services is a part of `CoreServices.framework` that lets you take advantage of remote programs on the Internet or a local intranet. Web services use standard protocols such as the Simple Object Access Protocol (SOAP) and XML-RPC to communicate service requests over HTTP. These standard protocols permit communication between otherwise disparate computing platforms and permit the creation of distributed services. The WebServicesCore framework defines an API for sending requests to remote servers using these protocols and for handling responses.

Application developers can also incorporate web service functionality into their applications using the Apple Event Manager and AppleScript. For more information, see *Inside Mac OS X: Making XML-RPC and SOAP Requests With AppleScript*.

Open Transport

Open Transport is a Carbon compatibility API in `CoreServices.framework` that provides legacy networking and communications support. Open Transport enables applications to use more than one networking system at once (for example, AppleTalk to communicate with network printers and TCP/IP to connect to the Internet). With Open Transport, users can save and modify different networking configurations and switch easily among them.

The version of Open Transport in Mac OS X supports the most commonly used interfaces in Mac OS 8 and Mac OS 9. For example, it supports the Open Transport endpoint routines for IP protocols. However, it does not include the connection-oriented transaction-based endpoint feature (which should affect only users of AppleTalk protocols such as ASP). Neither does it support the native XTI (X/Open Transport Interface) interfaces or BSD stream interfaces.

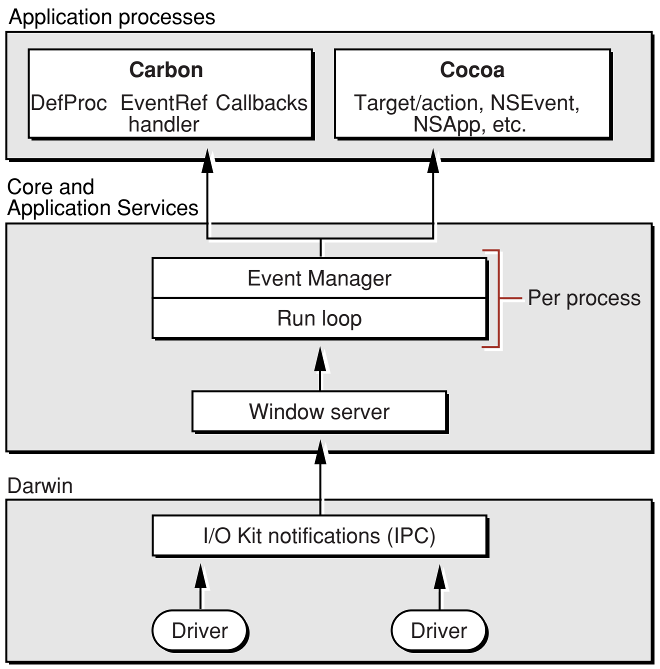
An important change from prior versions of Open Transport is the addition of client context parameters to a number of functions. Each client of Open Transport now has its own context so that Open Transport can track resources it allocates on behalf of the client. A client in this case is an application or a shared library, and resources are objects like endpoints, timer tasks, and blocks of memory.

Tracking a User Event

The perspective that Figure 3-1 (page 55) gives of Mac OS X as layers of system software suffices to illustrate the general interfaces and dependencies among parts of the system. But it does not adequately convey the dynamism of the operating system—in other words, how Mac OS X typically "works." An alternative approach to this static view of Mac OS X is one that follows a hypothetical user event through the system, from the click of a mouse to the handling of the event by the appropriate function or method in the appropriate application environment. It then traces, through the layers of the system, a hypothetical chain of events set off by the invocation of the function, resulting, in this case, in the drawing of a new object on the screen (say, a dialog).

Figure 3-7 depicts the environments and subsystems that generate, repackage, and forward an event along to its destination.

Figure 3-7 The handling of an event in Mac OS X


Application processes
Carbon
DefProc EventRef Callbacks handler
Cocoa
Target/action, NSEvent, NSApp, etc.
Core and Application Services
Event Manager
Run loop
Per process
Window server
Darwin
I/O Kit notifications (IPC)
Driver
Driver

A low-level event originates when the device driver that controls an input device such as the mouse or the keyboard detects a user action. The I/O Kit, which forms the foundation of all device drivers in Mac OS X, creates the event and puts it in the window server's event queue (see "Quartz Compositor" (page 69) for a discussion of the window server). This queue is in a block of memory shared by the I/O Kit and the window server. Once the I/O Kit puts an event in the queue, it notifies the window server via the Mach interprocess communication mechanism (IPC).

The window server then takes the event off the queue and consults a database of currently open windows. It sends the event to the event port of the run loop belonging to the process that owns the window where the event occurred. The Carbon Event Manager gets the event from the run-loop port, packages the event in an appropriate form, and passes it to the event-handling mechanism specific to the application environment of the process. This mechanism ensures that the event is handled by the function or method associated with the control that is clicked (or key that is pressed).

The event-handling mechanism is different for each application environment:

- **Carbon**. Carbon has several mechanisms that applications can use to handle events. The primary mechanism uses EventRefs, opaque low-level event structures. Handlers of EventRefs are installed on user-interface objects (including default ones by the Human Interface Toolbox), and these automatically receive all or some events destined for those objects. The handler can ignore the event, handle it, or pass it on to the next handler in the enclosing container. Event handling using DefProc messages and function callbacks is also possible.
- **Cocoa**. In Cocoa an event is packaged as an NSEvent object. The object is sent to the application object responsible for the overall management of an application process. The application object forwards the NSEvent object to the first-responder view in the window in which the event occurred. Through a next-responder mechanism, the event object, if not handled, can travel up the window's view hierarchy until it arrives at the application object itself. If the event is associated with a user-interface control, it is typically handled through a mechanism called "target-action."
- **Java**. Event handling in Java is implemented by the `java.awt.Event` and `java.awt.Component` classes.

For more information on how events between Carbon and Cocoa environments are handled, see *Inside Mac OS X: Integrating Carbon and Cocoa in Your Application*.

C H A P T E R 4

Booting and Logging In

This chapter describes what happens when a Mac OS X system boots up and when a user logs into the system. "Booting up" refers to the series of actions the system performs to prepare itself for use. This boot sequence includes a number of different tasks, including initializing hardware, starting system daemons, and displaying the login window. After a user logs in, the system completes an additional series of actions that sets up the computing environment for that user.

This chapter explains the aspects of the boot sequence that developers might encounter when trying to write their software. This chapter also explains how to use the various "hooks" available for customizing the booting procedures.

Note: Man pages, a form of online documentation, are available for most of the daemons and utilities mentioned in this chapter. You can consult these man pages for further information. To display a man page from the Terminal application, enter `man` on the command line followed by the name; for example, `> man getty` (where `>` is the prompt). You can also access man pages from the Help menu in Project Builder.

The Boot Sequence

From the moment a user turns on a Mac OS X system to the moment the login window appears, Mac OS X executes a boot sequence that readies the system for use. This section summarizes the events that happen during this sequence.

BootROM

When the power to a Macintosh computer is turned on, the BootROM firmware is the first code activated. BootROM (which is part of the computer's hardware) has two primary responsibilities: to initialize system hardware and to select an operating system to boot. BootROM has two components to help it carry out these functions:

- POST (Power-On Self Test) initializes some hardware interfaces and verifies that sufficient RAM memory is available and is in a good state.
- Open Firmware initializes the rest of the hardware, builds the initial device tree (a hierarchical representation of devices associated with the computer), and selects the operating system to use.

The same version of BootROM can start either Mac OS 9 or Mac OS X.

BootX

When BootROM (or the user) selects Mac OS X as the operating system to boot, control passes to the BootX booter (located in `/System/Library/CoreServices`). BootX's principal duty is to load the kernel environment. As it does this, BootX draws the "booting" image on the screen.

When loading the kernel environment, BootX first attempts to load a previously cached set of device drivers (called an mkext cache) for hardware that is involved in the boot process. If this cache is missing or corrupt, BootX searches `/System/Library/Extensions` for drivers and other kernel extensions whose `OSBundleRequired` property is set to a value appropriate to the type of boot (for example, local or network boot). See the kernel developer documentation for more on the `OSBundleRequired` key and the loading of device drivers during booting.

Once the kernel and all drivers necessary for booting are loaded, BootX starts the kernel's initialization procedure. At this point, enough drivers are loaded for the kernel to find the root device. Also from this point, Open Firmware is no longer accessible.

The kernel initializes the Mach and BSD data structures and then initializes the I/O Kit. The I/O Kit links the loaded drivers into the kernel, using the device tree to determine which drivers to link. Once the kernel finds the root device, it roots BSD off of it.

Finally, the kernel starts the `mach_init` process. The `mach_init` process is the Mach bootstrap port server, which enables Mach messaging.

After the root file system is mounted, system initialization proceeds to run the system startup items and launch any system daemons (see "System Initialization" (page 91)).

System Initialization

The `mach_init` process starts the BSD `init` process. This latter process, which has a process ID (PID) of 1, "owns" every other process on the system. Despite its centrality, the `init` process is simple. It performs four principal tasks:

1. It determines if the user wants single-user mode or is booting from a CD-ROM. If either of these conditions apply, an advisory is printed and control is handed over to the user.
2. It runs the system-initialization shell scripts—`/etc/rc.boot` and `/etc/rc`—which complete basic initialization tasks; for details, see "The rc.boot and rc Scripts" (page 92).

   The `/etc/rc` script runs the `SystemStarter` program, which handles more specialized initialization tasks specified as "startup items"; for details, see "Startup Items" (page 93).
3. Via the `getty` command, `init` launches the loginwindow application, which displays the login window and manages the user login procedure; for details, see "Managing User Sessions" (page 100).
4. As the parent of all processes, `init` handles all necessary cleanup of system processes as they terminate.

Although the `init` process owns every other process on the system, a distinction can still be made between user and system processes. Startup items and any applications run prior to the loginwindow application form the group of system processes. These applications provide services to all users of the system and are usually children of the `init` process. Processes created after the launching of loginwindow form the group of user processes. User processes are always associated with a particular user session and are usually children of the `Window Manager` process of the user's session.

Note: Not all user processes are children of the `Window Manager` process. Processes launched as root and some special system processes are owned by the user but are children of the `init` process. You can use the Process Viewer application to determine the owner and parent of any process on the system.

When a user logs out of a session, the loginwindow application terminates all processes running in the scope of that user session. This is a requirement for closing out the session cleanly. System processes are not affected by a user logout. System processes are terminated only when the system itself is shutdown or restarted. For more information on what happens during a logout, restart, or shutdown sequence, see "Logging Out" (page 102).

The rc.boot and rc Scripts

The `rc.boot` and `rc` Bourne shell scripts in `/etc` perform basic initialization tasks. First the `rc.boot` script performs a file-system consistency check (`fsck`) and synchronizes memory with the file system (`sync`). Then the `rc` script performs the following actions:

- It starts the device-driver loader (`kextd`).
- It starts the ATS server.
- It starts the window server.
- It runs the `update` background process, which periodically flushes the file-system cache.
- It creates the swap file for the virtual-memory system and starts the dynamic pager.
- As the final step, the `rc` script starts the `SystemStarter` program to process the local and system startup items (see "Startup Items" (page 93)).

Because the `rc` scripts are simple Bourne shell scripts, you can read the source to see exactly what takes place. For more information on the daemons started during system installation, see "System Daemons" (page 96).

Startup Items

Startup items are procedures that run during the last phase of booting to prepare a Mac OS X system for normal operation. They consist of programs (including shell scripts) that perform tasks such as clearing away temporary files and starting system daemons.

The system startup items provided by Apple are located in `/System/Library/StartupItems`. You should not modify the items in this directory. However, you can define your own startup items and store them in `/Library/StartupItems`. See "Creating Custom Startup Items" (page 105) for instructions on how to create your own startup items.

The `SystemStarter` program, which is the last thing run by the `rc` script, coordinates the execution of all startup items. To understand what `SystemStarter` does, it helps to understand what a startup item is. A startup item is a folder containing at least two files:

- a program (typically a shell script) whose name matches the folder name
- a configuration property-list file

The configuration property-list file for each startup item is named `StartupParameters.plist`. This property list specifies the name of the startup item and, more importantly, the dependencies for the startup item at multiple levels of granularity. The values in this property list indicate which services the startup item provides, which services must be run before the startup item, and which services the startup item uses (if available). `SystemStarter` uses this dependency information to determine the launch order for startup items.

The `SystemStarter` program launches as many startup items as possible in parallel. Startup items are organized into groups based on their dependencies. When the items from one group are launched, the items in the next group begin to launch.

For more on the key-value pairs in `StartupParameters.plist`, see "Creating Custom Startup Items" (page 105). For general information on property lists, see "Property Lists" (page 197).

Table 4-1 lists the core startup items provided with Mac OS X, along with a brief description of what they do. Other startup items may be installed by third-party applications or by system components such as QuickTime. The order given in the table is the *general* order in which the items are executed. The exact order is determined by the `SystemStarter` program at boot time.

Table 4-1 Core system startup items

Startup item	Description
SystemTuning	Sets up performance values for the system, based on such factors as available memory.
SecurityServer	Starts the security server, which oversees system authorization, authentication, and keychain access.
ConfigServer	Acts as a placeholder for startup items that depend on the `configd` daemon, which is now launched prior to the startup item phase. This startup item is deprecated and may be removed in a future release.
Disks	Starts the `autodiskmount` daemon, checks and mounts local disks.
Accounting	Starts up the accounting system (`accton`), which writes accounting information for each launched process to `/var/account/acct`.
Network	Configures the local network interfaces based on the data in `/etc/iftab`, sets the machine's host name, configures network routing (if specified), turns IP routing on or off (if specified), sets the machine's host ID in the kernel, and loads the Shared IP kernel extension to enable sharing of one IP address among all application environments (including Classic).
NetworkExtensions	Loads several network-related kernel extensions.
BIND	Starts up the local DNS-server.
mDNSResponder	Starts up the multicast-DNS responder, which supports Rendezvous, Apple's implementation of zero-configuration networking.
Portmap	Starts up the `portmap` daemon.

Table 4-1 Core system startup items (continued)

Startup item	Description
SystemLog	Starts the system log daemon (`syslogd`).
AppleTalk	Runs the `AppleTalk` startup program in either router mode, multihoming (nonrouter) mode, or on a single port (as defined in `/etc/hostconfig`).
CrashReporter	If crash reporting is enabled, starts the crash reporting system (`CrashReporter`).
LDAP	Starts the LDAP service.
NetInfo	Starts the NetInfo servers (`netinfod` and `nibindd`).
NIS	Starts the Network Information Services.
DirectoryServices	Starts the name-resolver daemon (`lookupd`)
NetworkTime	Starts up network time services, which uses the Network Time Protocol (NTP).
LoginWindow	Signals that the system has the services it needs to start the loginwindow application. The `init` process launches the application while `SystemStarter` continues loading startup items in the background.
SSH	If remote access is enabled, starts the `sshd` daemon. This startup item also generates the secure shell encryption keys as needed.
NFS	Starts the Network File System service that performs asynchronous block I/O (`nfsiod`), mounts remote file systems, starts the automounter, and, if the NetInfo database indicates that the computer should export a file system using NFS, starts the NFS server.
Samba	Starts the Samba server (`smbd`), which provides network connectivity via the Server Message Block (SMB) protocol.
AuthServer	Starts the authentication server.
AppServices	Starts the core services daemon (`coreservicesd`).
PrintingServices	Starts the CUPS printing services daemon (`cupsd`).

Table 4-1 Core system startup items (continued)

Startup item	Description
IPServices	Starts TCP/IP services (`inetd`), host configuration services (BOOTP), and the netboot client management server.
AppleShare	If a network connection is detected, starts the AppleShare service.
SendMail	Starts the outgoing mail services daemon (`sendmail`) and cleans up the outbound mail queue directory.
Apache	If web sharing is enabled, runs the Apache HTTP server `httpd`.
Cron	Runs the `cron` daemon.

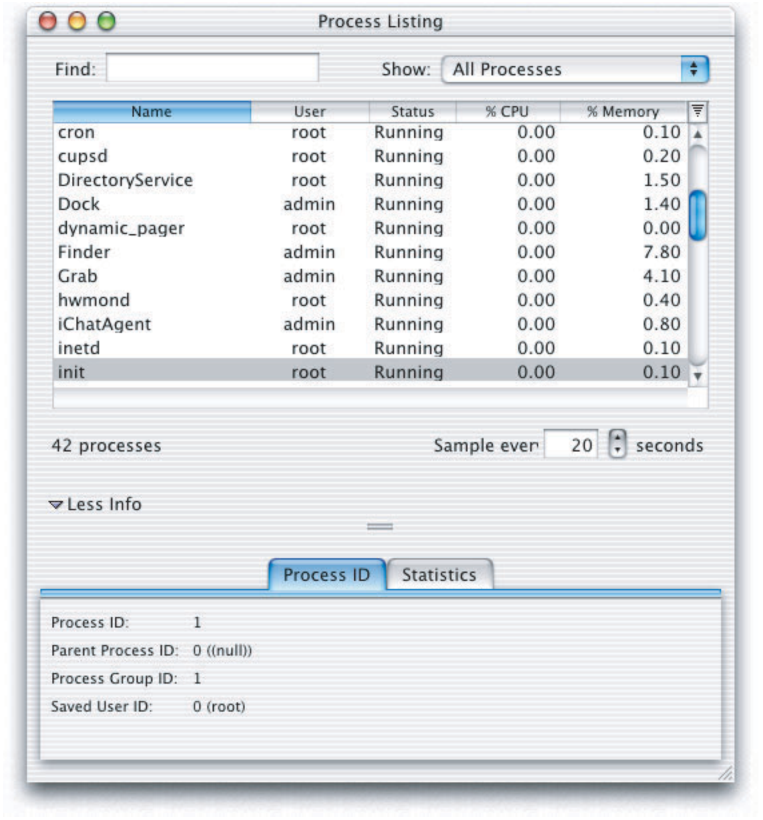
The section "System Daemons" (page 96) briefly describes some of the daemons and services mentioned in this table.

The existence of a startup item in the `/System/Library/StartupItems` folder does not guarantee the availability of that item's services to the user. Each startup item has access to the configuration information in `/etc/hostconfig` and can use that information to assess whether or not to launch its services. Many startup items use this information, although some required startup items do not. Among the startup items that check this configuration information are AppleTalk, AppleShare, AuthServer, Disks, SSH, SendMail, IP Services, and Apache.

System Daemons

By the time a user logs in to a Mac OS X system, a number of processes are already running. Most of these processes are daemons or servers created by the system and running in the background. A handful are created on behalf of the user by the loginwindow application and the `Window Manager` daemon. Launching the Process Viewer (located in `/Applications/Utilities`) immediately after login results in a window similar to the one shown in Figure 4-1.

Figure 4-1 Processes shown in Process Viewer



Name	User	Status	% CPU	% Memory
cron	root	Running	0.00	0.10
cupsd	root	Running	0.00	0.20
DirectoryService	root	Running	0.00	1.50
Dock	admin	Running	0.00	1.40
dynamic_pager	root	Running	0.00	0.00
Finder	admin	Running	0.00	7.80
Grab	admin	Running	0.00	4.10
hwmond	root	Running	0.00	0.40
iChatAgent	admin	Running	0.00	0.80
inetd	root	Running	0.00	0.10
init	root	Running	0.00	0.10

42 processes Sample ever 20 seconds

Process ID:	1
Parent Process ID:	0 ((null))
Process Group ID:	1
Saved User ID:	0 (root)

Table 4-2 describes some of the standard daemons and servers. Depending on the system configuration and user preferences, some of these daemons may be disabled or set up to launch on demand (see "Launching Daemons on Demand" (page 99)). Conversely, your system may have other daemons running because of additional installed services.

Table 4-2 Common system daemons and servers

Daemon	Description
`ATSServer`	The Apple Type Solution server, enabling system-wide font management.
`autodiskmount`	Automatically mounts all devices, including hard disks, and removable media.
`automount`	Automatically mounts NFS file systems when they are first accessed and later unmounts them when they are idle. A mount point for a virtual file system first appears as a symbolic link on a local file system. Reading this symbolic link triggers `automount` to mount the associated remote file system.
`configd`	Automatically configures and maintains the network.
`cron`	Executes scheduled commands or scripts.
`dynamic_pager`	Communicates with the kernel's default pager to create or delete the swap files (in `/private/var/vm`); these files are used as backing store for virtual memory.
`inetd`	An Internet "super-server" that listens for connections on certain sockets. When a connection occurs, it decides what service the socket corresponds to and invokes the appropriate program to service the request.
`init`	The system initialization process. This process is the parent for most system-level processes.
`kextd`	The device-driver loader. During system startup, this daemon loads any remaining device drivers that were not loaded by BootX. After startup, `kextd` is responsible for dynamically loading and unloading device drivers.
`lookupd`	A name resolver, which expedites look-up requests to network information services such as NetInfo and DNS.

Table 4-2 Common system daemons and servers (continued)

Daemon	Description
`mach_init`	The Mach bootstrap port server, through which Mach messaging is enabled. It also starts the `init` process.
`mDNSResponder`	The multicast-DNS responder daemon.
`netinfod`	NetInfo servers, one for each domain served.
`nfsiod`	Services asynchronous requests to an NFS server. Most systems have multiple instances of this daemon running.
`nibindd`	Finds, creates, and destroys NetInfo servers (see `netinfod`).
`pbs`	The pasteboard server (similar to the Clipboard in Mac OS 9) enables the exchange of data between applications. It is also the data-transfer mechanism used in dragging operations. See "The Clipboard" (page 79) for more information.
`portmap`	Converts RPC program numbers into Internet (DARPA protocol) port numbers. It must be running before RPC calls can be made.
`syslogd`	Logs system error and status messages.
`update`	Periodically flushes the file-system cache to help prevent data loss in the event of a crash.
`Window Manager`	The window server. See "Quartz Compositor" (page 69) for more information. This process is the parent for most user processes.

For further information consult the man pages available for most of these daemons.

Launching Daemons on Demand

Mac OS X provides support for on-demand launching of system daemons and uses this feature for some system startup items. On-demand launching improves system performance by delaying the launch of any daemons that are not needed immediately. During periods of inactivity, daemons can also shut down to conserve system resources.

On-demand launching is transparent to clients of the services that support it. When a client sends a message to a daemon that is currently shut down, `mach_init` automatically loads any services required by the daemon and then launches the daemon. Once the daemon is up and running, it checks in with `mach_init` and asks for the receive rights to its Mach ports. Once it has the receive rights for its ports, the daemon can respond to messages on those ports normally.

How to create an on-demand daemon is beyond the scope of this book because it involves interactions with the kernel. For more information on the kernel and Mach-port messaging, see *Inside Mac OS X: Kernel Programming*.

Managing User Sessions

During the system initialization cycle, the `init` process launches the loginwindow application to manage user sessions. The loginwindow application displays the login window and provides several services on behalf of the user. This section discusses those services and the user environment provided by loginwindow.

Logging In

The loginwindow application coordinates the login process and user session, calling on other system services as needed. Depending on the user's login preferences, loginwindow may prompt the user for a valid login name and password or use cached values to log the user in automatically. When the user's login name and password have been authenticated (see "Authenticating Users" (page 101)), loginwindow proceeds to load the user environment.

When a user logs in, loginwindow does the following:

- Loads the user's computing environment (including preferences, environment variables, device and file permissions, keychain access, and so on). See "Setting Up the User Environment" (page 102) for more information.
- Launches the Dock, Finder, and SystemUIServer.
- Launches the Setup Assistant if an installation is in progress.

- Automatically launches applications specified in the Login Items System Preferences.

When all of the specified applications are launched and running, the login procedure is complete.

The loginwindow application uses Launch Services to launch all applications, including the Finder, Dock, SystemUIServer, and user-specified applications. Most applications in the user session run as child processes of the `Window Manager` process, not loginwindow.

If the Finder or Dock processes die for some reason, loginwindow automatically launches them again. In the same vein, if the loginwindow application dies, the `init` process automatically restarts it.

Once the user session is up and running, loginwindow monitors the session and user applications in the following ways:

- Manages the logout, restart, and shutdown procedures. See "Logging Out" (page 102) for more information.
- Manages the Force Quit window, which includes monitoring the currently active applications and responding to user requests to force-quit applications and relaunch the Finder. (Users open this window from the Apple menu or by pressing Command-Option-Escape.)
- Displays alert dialogs upon receiving notifications from hidden applications (that is, applications with no visible user interface).
- Writes any standard-error (`stderr`) output to a log file (`/var/tmp/console.log`), which is then used as input by the Console application.

Authenticating Users

Mac OS X requires user authentication prior to accessing the system. Although the loginwindow application manages the user authentication process, it does not authenticate the user itself. The loginwindow application takes the user information specified in the login screen and passes it to Directory Services for authentication. As soon as Directory Services authenticates the user, loginwindow initiates the user session.

The loginwindow application does not prompt the user for login information if the "Enable automatic log in" option is selected. Only one user at a time may enable this option from the Users pane of the Accounts System Preferences. From this same pane, the user must also enter a valid password by clicking the Set Auto Login control.

Setting Up the User Environment

When a user logs in to the system, the loginwindow application sets up the user environment and records information about the login. The following list outlines the tasks performed by loginwindow to restore the user session:

1. Secures the login session from unauthorized remote access. Applications that are launched remotely are not registered with the pasteboard server's (that is, the Clipboard's) port. As a result, some standard features are blocked for these processes, including copy, cut, paste, Apple events, window minimization, and other services.
2. Records the login in the system's `utmp` database.
3. Sets owner and permissions for the console terminal.
4. Resets the user's preferences to include global system defaults (NSRegistrationDomain).
5. Registers the pasteboard server (`pbs`) with the bootstrap port and launches `pbs`.
6. Configures the mouse, the keyboard, and system sound using the user's preferences.
7. Sets the user's group permissions (`gid`).
8. Retrieves the user record from Directory Services and applies that information to the session.

Logging Out

The procedures for logging out, restarting the system, or shutting down the system have similar semantics. The foreground process usually initiates these procedures in response to the user choosing an item from the Apple menu; however, a process can also initiate the procedure programmatically by sending an appropriate Apple event to the loginwindow application. The loginwindow application carries out the procedure, posting alerts and notifying applications to give them a chance to clean up before closing.

A typical logout/restart/shutdown procedure proceeds as follows:

1. The user selects Log Out, Restart, or Shut Down from the Apple menu.
2. The foreground application initiates the user request by sending an Apple event to the loginwindow application. (See "Application Responsibilities" (page 103) for a list of events.)
3. The loginwindow application displays an alert to the user asking for confirmation of the action.
4. If the user confirms the action, loginwindow sends a Quit Application Apple event (`kAEQuitApplication`) to every foreground and background user process.
5. The loginwindow application closes out the user session and continues with the action.

   - On a logout action, loginwindow dequeues all events in the event queue, starts the logout-hook program (if one is defined), records the logout, resets device permissions and user preferences to their defaults, and returns control to the display of the login window. (See "Customizing Login and Logout Procedures" (page 110) for more on the logout hook.)
   - On a restart, loginwindow sets the device permissions and user preferences to their defaults, powers off the system, and then powers it back on to start the boot process.
   - On a shutdown, loginwindow powers off the system.

Foreground processes can choose not to terminate when they receive the Quit Application event. See "Terminating Processes" (page 104) for more information.

Application Responsibilities

To initiate a logout, restart, or shutdown sequence, the foreground application must send an appropriate Apple event to the loginwindow application. Upon receipt of the event, the loginwindow application begins the process of shutting down the user session. Depending on the Apple event sent by the process, loginwindow may or may not notify the user with an alert and give the user a chance to abort the sequence.

The following list shows the preferred Apple events for logout, restart, and shutdown procedures. These events have no required parameters.

- `kAELogOut`
- `kAEShowRestartDialog`
- `kAEShowShutdownDialog`

Upon receipt of one of these events, loginwindow posts an alert notifying the user of the impending action. At this point, the user may continue with the action or abort it. If the user continues with the action, loginwindow sends an Apple event to each application asking it to quit. See "Terminating Processes" (page 104).

In addition to the preferred Apple events, there are two additional events that tell loginwindow to proceed immediately with a restart or shutdown sequence. The `kAERestart` and `kAEShutDown` Apple events proceed with a polite restart or shutdown of the system. However, these events should be used judiciously because neither of them posts a message to the user or allows the user to abort the sequence.

Important
Note that if a logout, restart, or shutdown event originates from an application in the Classic environment, the event affects only the Classic environment and its applications. The rest of the user session continues running.

Terminating Processes

Prior to carrying out a log out, restart, or shutdown sequence, the loginwindow application attempts to terminate all foreground and background user processes. The loginwindow application sends each process a `kAEQuitApplication` Apple event to give it a chance to shut down gracefully. For foreground processes, loginwindow then waits for a reply. For background processes, loginwindow terminates the process, by sending a kill command, regardless of any reply.

When a foreground process receives the Quit Application Apple event (`kAEQuitApplication`) from loginwindow, it should terminate itself immediately or post an alert dialog when a user decision is required first (such as when there is an unsaved document); when that condition is resolved the application should then terminate. If the user decides to abort the termination sequence (by clicking Cancel in a Save dialog, for example) the application should respond to the event by returning a `userCancelledErr` error.

If a foreground application fails to reply or terminate itself after 45 seconds, loginwindow automatically aborts the termination sequence. This safeguard is to protect data in various situations, such as when an application is saving a large file to disk and is unable to terminate in the allotted time. If a foreground application is unresponsive and not doing anything, the user must use the Force Quit window to kill it before proceeding.

For user background processes that link with Carbon, Cocoa, or Java, the procedure is a little different. Loginwindow notifies the process that it is about to be terminated by sending it a Quit Application Apple event (`kAEQuitApplication`). However, it does not wait for a reply from the process. Loginwindow proceeds to terminate any open background processes, regardless of any returned errors.

The loginwindow application does not terminate processes residing in the system context. This rule applies to all daemons and system startup items that are launched by the system at boot time. These processes reside outside the context of the user session and are terminated only during a restart or shutdown sequence. Loginwindow also does not kill background processes that are independent of Carbon, Cocoa, or Java, even if they are launched from the user context. (Though launched from a user context, these processes are taken over by the system when the user logs out.) Mac OS X does not send any notifications to system processes before terminating them.

Customization Techniques

The Mac OS X boot process includes several entry points that developers can use to customize the process. Developers can create new system daemons and services to be launched at boot time. Developers and system administrators can also create initialization scripts to be run when a user logs in to the system.

Creating Custom Startup Items

You can add specialized behavior to the booting sequence by creating new startup items. A startup item, as described in "Startup Items" (page 93), is a program or shell script that performs services such as launching system daemons, deleting old files, or any number of other maintenance tasks prior to the first login session.

Startup items are run (through the `SystemStarter` program) as the final phase of the booting sequence. At that time, `SystemStarter` looks for startup items in the `/System/Library/StartupItems` and `/Library/StartupItems` directories. It gathers information from the property list of each startup item and uses that information to determine the execution order for the items. It then executes the startup items in groups based on the dependencies of the items.

The `/System/Library/StartupItems` directory is reserved for startup items that ship with Mac OS X. All other startup items should be placed in the `/Library/StartupItems` directory. Note that this directory does not exist by default and may need to be created during installation of the startup item.

To create a new startup item, you must create a program or script to execute your code, and you must create a property list file to contain information about your startup item. The following sections describe these techniques in more detail.

Note: Although a single startup item can perform multiple tasks and launch multiple daemons, it is recommended that you limit your startup items to one task. Implementing one task per item makes it easier to isolate dependencies and to speed up the loading of startup items.

Creating the Startup Item Executable

The startup item executable contains your startup item's code and can be implemented as a command-line executable or shell script. Normally, your executable is called only at system startup time. However, SystemStarter does support the ability to call your executable at a later time (see "Starting and Stopping Startup Items" (page 109)).

To create a startup item:

1. Create a directory for your startup item. The directory name should correspond to the behavior you're providing.

   Example: `MyDBServer`

2. Add your executable to the directory. The name of your executable should be exactly the same as the directory name.

   Example: `MyDBServer/MyDBServer`

3. Create the property list with the name `StartupParameters.plist` and add it to the directory. See "Specifying the Startup Item Properties" (page 107).

Example: `MyDBServer/StartupParameters.plist`

4. Create an installer to place your startup item in the `/Library/StartupItems` directory of the target system. (Your installer may need to create this directory first.)

Mac OS X provides some shell script code to simplify the process of creating startup items. The file `/etc/rc.common` defines some routines that are useful for processing command-line arguments and for gathering system settings. To use these routines, source the `rc.common` file in a shell script and call the `RunService` routine, passing it the first command-line argument, as shown in the following example:

```
#!/bin/sh
. /etc/rc.common
#
# Your startup item code
#

RunService "$1"
```

The `RunService` routine looks for `StartService`, `StopService`, and `RestartService` routines in your shell script and calls them to start, stop, or restart your services as needed. You must provide implementations for all three routines, although the implementations can be empty for routines whose commands your service does not support.

If your startup item executable contains code that might take a while to complete, you should consider running that code as a daemon or a background process. Executing lengthy startup tasks directly from your scripts delays system startup. Your startup item script should execute as quickly as possible and then exit.

Specifying the Startup Item Properties

In addition to the startup item executable, every startup item must have a property list with the name `StartupParameters.plist` in the startup item directory. This property list contains information about the startup item as well as any dependencies this item has on other startup items. Prior to launching any startup items, the `SystemStarter` program reads the property list of each startup item and uses the dependency information to determine the launch order for startup items.

Table 4-3 lists the key-value pairs to include in the `StartupParameters.plist` file of your startup item. All arrays and dictionaries contain string values. You can use the Property List Editor application in `/Developer/Applications` to create this property list.

Table 4-3 `StartupParameters.plist` key-value pairs

Key	Type	Value
Description	String	A short description of the startup item, used by administrative tools.
Provides	Array	The services provided by this startup item. Although a startup item can provide multiple services, it typically provides only one.
Requires	Array	The services provided by other startup items that must be run before this startup item can be started. If the required services are not available, this startup item is not run.
Uses	Array	The services provided by other startup items that should be started before this startup item, but which are not mandatory. The startup item should be able to start without the availability of these services.
OrderPreference	String	For those startup items with the same execution order (as determined by the Requires and Uses values), the relative order in which they should be started. There are five order-preference values: `First`, `Early`, `None`, `Late`, and `Last`. The default is `None`. The order preference is an advisory value and might be ignored.

The values you specify for the Requires and Uses properties correspond to the service name on which your startup item is dependent. Service names may not always correspond directly to the name of the startup item that provides that service. The Provides property specifies the names of the services provided by a startup item, and while this name usually matches the name of the startup item, it is not required to do so. In particular, startup items that launch multiple services can have at most one service whose name matches the name of the startup item.

If two startup items provide a service with the same name, SystemStarter runs only the first startup item it finds with that name. For this reason, it is recommended that your startup items do not provide multiple services unless absolutely required by codependencies in your startup code.

The values of the Requires, Uses, and OrderPreference keys do not guarantee a particular launch order.

Starting and Stopping Startup Items

When SystemStarter executes a startup item, it passes the executable an argument indicating the desired action. On startup, SystemStarter passes the executable a start argument. After startup, SystemStarter can also pass restart or stop parameters to a startup item. Your startup item executable should check the supplied argument and act accordingly to start, restart, or stop the corresponding services.

If you want to start, restart, or stop startup items from your own scripts, you can do so using the SystemStarter program. To use SystemStarter, you must execute it with two parameters: the desired action and the full path to the target startup item directory. For example, to restart the Apache Web server, you would type the following at the command prompt:

```
/sbin/SystemStarter restart /System/Library/StartupItems/Apache
```

Important
You must have root authority to start, restart, or stop startup items.

Startup items should always respect the arguments passed in by SystemStarter. However, the response to those arguments is dependent on the startup item. The stop and restart options may not make sense in all cases. Your startup item could also support the restart option using the existing code for stopping and starting its service.

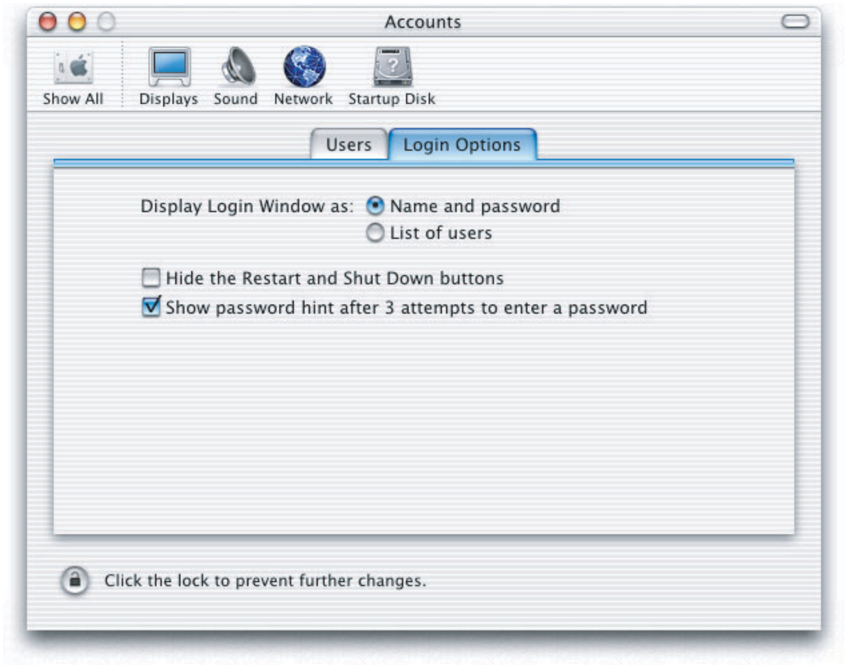
Applications that need to manipulate startup items at runtime can also do so using the SystemConfiguration framework. This framework supports getting references to a desired startup item and starting it.

For more information about startup items, see "Creating Custom Startup Items" (page 105).

Customizing Login and Logout Procedures

Administrators can already configure several features of loginwindow through the Login pane of System Preferences. The Login Options pane of the Accounts System Preferences (shown in Figure 4-2) shows some additional options for administrators of the computer.

Figure 4-2 The Login Options pane of Accounts system preferences



Display Login Window as: [✓ Name and password] List of users

☐ Hide the Restart and Shut Down buttons

☑ Show password hint after 3 attempts to enter a password

Click the lock to prevent further changes.

In addition to these features, the loginwindow application provides a set of programmatic hooks that you can use to customize the login procedure. You can use these hooks to run additional programs or scripts on each login or logout. There is also a hook to disable the Restart and Shut Down buttons, similar to the option available in the Login Options pane.

To use the loginwindow hooks, you must modify the code for launching loginwindow found in `/etc/ttys`. In this file is the following line:

```
console "/System/Library/CoreServices/loginwindow.app/loginwindow" vt100 on
secure window=/System/Library/CoreServices/WindowServer
onoption="/usr/libexec/getty std.9600"
```

This line tells the `init` program to launch loginwindow on the console terminal and to use `WindowServer` (which is a symbolic link to the `Window Manager` process) as the windowing-system process. Into this line, you can insert additional parameters for loginwindow to process. Table 4-4 lists the parameters currently supported by loginwindow.

Table 4-4 loginwindow parameters

Parameter	Description of value
`-LoginHook`	The full path to a script or tool to run when a user successfully logs in.
`-LogoutHook`	The full path to a script or tool to run when a user successfully logs out.
`-PowerOffDisabled`	If “YES,” the Shut Down and Restart buttons in the login window are disabled; also, pressing the computer’s power button quits the Finder and Dock applications but does not turn off the system. This feature prevents users from casually powering down a system that provides some shared service, such as a print server or file server.

The `-LoginHook` and `-LogoutHook` parameters are particularly useful because they permit custom administrative, accounting, or security programs to run as part of the login and logout procedures. For example, your modified console definition in `/etc/ttys` might look similar to the following:

```
console "/System/Library/CoreServices/loginwindow.app/loginwindow
-PowerOffDisabled YES -LoginHook
/Users/Administrator/Scripts/mailLoginToAdmin" vt100 on secure
window=/System/Library/CoreServices/WindowServer
onoption="/usr/libexec/getty std.9600"
```

Using Environment Variables

Mac OS X supports UNIX-style environment variables on a per-user basis. On login, the loginwindow application looks for a special property list file with the name `environment.plist`. This file must be located in a directory called `.MacOSX` at the root of the user's home directory. The path to this file is as follows:

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

If an `environment.plist` file exists, loginwindow reads the keys from the file and registers them as user environment variables. This file supports only the definition of environment variables. You cannot use this file to execute other forms of script code. The format of the file is the same XML format as other property list files, with each key in the file containing a string value.

Replacing the Finder

Mac OS X supports the ability to launch programs other than the Finder system application upon login. This feature is intended for users who want a different default application to be launched at login time. For example, you could launch the Terminal application at login to display a command-line interface instead of the Finder.

To replace the Finder system application, use the `defaults` utility to modify the loginwindow preferences. The loginwindow preferences are stored in `com.apple.loginwindow`. The key you want to modify is the `Finder` key. Use the following syntax to write a new value for this key, specifying the full path to the replacement application:

```
defaults write com.apple.loginwindow Finder path
```

Users can execute this command from the Terminal application to change their own default settings. To change this setting for all users, you must execute this command from Terminal as root. You could also modify the `/etc/ttys` script or create your own login hook to execute the hook at each user login.

Using this command does not prevent users from launching the Finder manually once they have logged in. It does prevent loginwindow from automatically relaunching the Finder in the event of a crash or the user killing it from the Force Quit window.

For more information on the `defaults` utility, see "The defaults Utility" (page 206) in the chapter "Software Configuration" and consult the `defaults` man page.

Bundles

A bundle is a directory in the file system that stores executable code and the software resources related to that code. (It can contain only executable code or only software resources, but that is unusual). The bundle directory, in essence, "bundles" a set of resources in a discrete package. The resources include such things as images, sounds, and localized character strings that are used by some piece of software. Because code and associated resources are in one place in the file system, installation, uninstallation, and other forms of software management are easier.

Applications, frameworks, and loadable bundles (including plug-ins) are types of bundles. Internally, the structure of these bundle types is (or can be) quite similar. What primarily differentiates applications, frameworks, and loadable bundles are the characteristics and purpose of the executable code they contain. Each of these types has its own required extension: `.app`, `.framework`, and `.bundle` (or whatever extension is application-defined for a loadable bundle).

In a program, bundles are represented by programmatic entities such as instances of a class or (in procedural languages) objects of opaque types. Routines of these entities make bundle resources available to the program code that requests it. Other routines enable you to load and link executable code into a running application. Applications can load the code in loadable bundles whenever they need that code. Frameworks automatically—and dynamically—load and link shared library code.

Bundles can contain multiple sets of resources, each set of which groups resources by language, locale, and platform. By combining these sets of resources and executable images into a single package, you can create one version of your application, framework, or plug-in that executes properly on any supported platform. Using this model, you can automatically localize an application's human interface according to the user's language preferences.

Typically the Finder displays a bundle directory to users as a file to avoid unwarranted tampering with the bundle's contents. But the directory structure of some bundles, such as frameworks, is not hidden. Whether the Finder displays a bundle as a file or folder depends on several factors, including whether the bundle bit—a Finder attribute—is set in the bundle directory. Finder also hides the extensions from all application bundle names.

Note: Frameworks in the current release of Mac OS X are "versioned" bundles, because their different internal structure reflects their scheme for versioning dynamic shared libraries. This structure lacks many of the features of the newer types of bundles. See the chapter "Frameworks" (page 139) for more information on these types of bundles.

Benefits of Using Bundles

Bundles provide a variety of important advantages over the traditional Mac OS 9 software packaging scheme.

- A single bundle executable can run on Mac OS 9 and Mac OS X.
- A single bundle can support multiple chip architectures (PowerPC, x86), library architectures (CFM, Mach-O), and other special executables (for example, optimized libraries for the Velocity Engine).
- A single bundle can support multiple languages through an internationalization architecture. You can easily add new localized resources or remove unwanted ones.
- Bundles can reside on volumes of many different formats, including multiple fork formats like HFS, HFS+, and AFP, and single-fork formats like UFS, SMB, and NFS.
- You can index and access Help files and other bundle information resources through Sherlock.
- You can install, relocate, and remove bundles simply by dragging them.

Versioned bundles, described in the next section, "Anatomy of a Bundle" (page 117), do not share the first two features in the above list, namely support for multiple chip architectures and an executable that can run on the various Mac OS systems.

Anatomy of a Bundle

Bundles contain executable code and can contain a variety of resources such as

- images
- sounds
- localized character strings
- Resource Manager–style resource files
- libraries and frameworks
- plug-ins and other loadable bundles
- archived user-interface definitions

Mac OS X supports two different layouts for bundle directories, "new-style" and versioned. The directory layout for versioned bundles is inherited from Mac OS X's predecessor operating systems. The following example depicts this layout:

```
MyBundle.bundle/
    MyBundle (executable code)
    Resources/
        Pretty.tiff (nonlocalized resource)
        English.lproj/ (localized resources)
            Stop.eps
            MyBundle.nib
            MyBundle.strings
        French.lproj/ (localized resources)
            Stop.eps
            MyBundle.nib
            MyBundle.strings
```

Although the newest development tools on Mac OS X create only new-style bundles (with the exception of frameworks), the system bundle routines can read and manipulate both styles of bundles.

The remainder of this section describes the layout of new-style bundles, explaining where the executable code and resources go within a bundle. On disk, a bundle exists as a directory hierarchy. Minimally, a bundle has the structure shown in Listing 5-1:

Listing 5-1 A minimal bundle

```
- MyBundle/
    Contents/
        Info.plist
```

In other words, the `Contents` directory and, inside it, the `Info.plist` file must be present in a bundle. These files are important to how the bundle is treated by Finder and other parts of the operating system. They describe the bundle's various attributes.

Note: Use of the `PkgInfo` file is deprecated and is no longer required for bundles. You may still include this file in your bundles, but type and creator information should also be stored in the bundle's `Info.plist` file.

The information property list, `Info.plist`, contains key-value pairs stored in XML format. These pairs specify attributes such as the name of the bundle, the name of the bundle executable, version information, type and creator codes, application and document icons, and other metadata. System routines allow the bundle executable to read these attributes at runtime. In addition to the default bundle attributes, subsystems may place their own attribute information in the `Info.plist` file for easy access at runtime. You are free to store any application-defined data in the information property list as well. See "Information Property Lists" (page 198) in the chapter ""Software Configuration" (page 197)" for more on information property lists, including an example of one.

A special localized resource file named `InfoPlist.strings` goes with the `Info.plist` file. The former file contains keys for the information property list that need to be localized such as the CFBundleName key. For more information on localizing bundle names, see "Localizing Bundle Names" (page 177).

From the minimal bundle layout, a bundle directory can expand to a fully fleshed-out bundle such as you might find in a complex application. Listing 5-2 shows what might go into such a bundle.

Listing 5-2 The bundle layout of a complex application

```
- MyBundle/
    MyApp /* alias to Contents/MacOSClassic/MyApp */
    Contents/
        MacOSClassic/
            MyApp
            Helper Tool
        MacOS/
            MyApp
            Helper Tool
        Info.plist
        Resources/
            MyBundle.icns
            Hand.tiff
            Horse.jpg
            WaterSounds/
            en_US.lproj/
                MyApp.nib
                bird.tiff
                Bye.txt
                house.jpg
                house-macos.jpg
                house-macosclassic.jpg
                InfoPlist.strings
                Localizable.strings
                CitySounds/
            en_GB.lproj
                MyApp.nib
                bird.tiff
                Bye.txt
                house.jpg
                house-macos.jpg
                house-macosclassic.jpg
                InfoPlist.strings
                Localizable.strings
                CitySounds/
            Japanese.lproj/
                MyApp.nib
                bird.tiff
                Bye.txt
                house.jpg
                house-macos.jpg
                house-macosclassic.jpg
                InfoPlist.strings
                Localizable.strings
                CitySounds/
        Frameworks/
        PlugIns/
        SharedSupport/
```

Although there are different types of bundles, they all share certain features. At the top level of the bundle there is always a `Contents` directory. The `Resources`, `Frameworks`, `SharedSupport` and `PlugIns` directories are optional and appear only as necessary.

Important
You should avoid hard-coding directory paths to items within bundles because the internal structure of bundles could change. Instead, use the appropriate bundle APIs provided by Apple.

Several directories contain, as their names suggest, executable code for specific platforms. When a bundle's code is requested, the system searches for code in the format appropriate to the underlying operating system. The names of the platform-specific executable directories are `MacOSClassic` and `MacOS`. The name of the executable file inside these directories is typically the same name as the bundle name (minus the extension).

Resources can be localized or nonlocalized—that is, suitable for all localizations. Each set of localized resources goes into its own directory in the bundle. The `Resources` directory contains resource files grouped according to a language and, possibly, a region where a variant of that language is spoken. These directories have the extension of `.lproj` (the "l" stands for "language"). Each such directory contains all localizable resources for a particular language, and often for region-specific versions of that language. Nonlocalized resources are put in the level directly above the `.lproj` directories as there need be only one version of these files. One of these nonlocalized resources is the icon file for the bundle (which is the application icon if it's an application bundle). By convention, this file takes the name of the bundle and an extension of `.icns`; the image format can be any supported type, but if no extension is specified, the system assumes `.icns`. See "Localized Resources" (page 125) for more on bundle resources.

A bundle often stores each resource in its own file instead of grouping them in a single file, as does the Resource Manager. Localizable strings, however, are stored together in a "strings" file (so called because it has an extension of `.strings`). The reason for storing localizable strings in one file is that the contents can then be easily cached for better performance.

For more information on bundle resources, including localized strings, see the chapter "Internationalization" (page 209).

Important
Apple recommends that you do not package resources in the resource fork of the bundle's executable files.

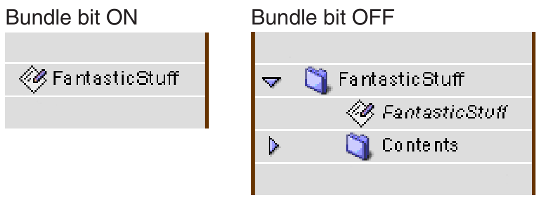
The `Frameworks` directory contains frameworks that are inextricably bound to the application. These dynamic shared libraries of these frameworks are revision-locked and will not be superseded by any other, even newer, versions that may be available to the operating system.

The `Frameworks`, `PlugIns`, and `SharedSupport` directories occur mostly in application bundles. See the chapter "Application Packaging" (page 131) for further information on these directories.

The Finder and Bundles

When you create a bundle, the build system can set a Finder attribute called a "bundle bit" in the bundle folder. Before the Mac OS X Finder displays a bundle in one of its windows, it reads this attribute. If the bundle bit is turned on, the bundle appears as a file package. A file package is a folder that the Finder presents to users as if it were a file (see Figure 5-1). In other words, the Finder hides the contents of the folder from users. This opacity discourages users from inadvertently (or intentionally) altering the contents of the bundle.

Figure 5-1 The Finder's bundle bit



Some bundles might not have the bundle bit set; this is the case with Apple-provided bundles. Yet the Finder can still handle them appropriately. As explained in the next section ("Types of Bundles" (page 122)), bundle folders should have extensions indicating their type—`.app`, `.framework`, `.bundle`, and so on. When the Finder encounters one of these folder extensions and determines that the folder is indeed a bundle, it does the proper things:

- Except for frameworks, it displays the bundle as a file package.

  Frameworks are displayed as folders so that you can browse their header files.

- If the bundle is an application (also known as an application package), Finder hides the `.app` extension.

- It extracts or computes the runtime information it needs from the bundle (type and creator codes, for instance) and updates its databases with it.

For more information on the Finder and how it handles bundles and documents, see the chapter "The Finder" (page 187).

Types of Bundles

Mac OS X recognizes at least three distinct types of bundles:

- **Application**. For Mac OS X applications, the application package is a bundle that contains the resources needed to launch the application, including the application executables.

- **Framework**. A framework is a bundle containing a dynamic shared library and all the resources that go with that library, such as header files, images, and documentation.
- **Loadable bundle**. Like an application, a loadable bundle usually contains executable code and associated resources. Loadable bundles differ from applications and frameworks in that they must be explicitly loaded into a running application. There are some special types of loadable bundles, two of which are especially noteworthy.
  - **Palette**. A palette is a type of loadable bundle specialized for Interface Builder. It contains custom user-interface objects and compiled code that are loaded into an Interface Builder palette.
  - **Plug-in**. A plug-in is a special type of loadable bundle that requires an architecture and an implementation above and beyond the simple code-loading and function-lookup functionality of the regular bundle programming interfaces.

  In addition, kernel extensions (KEXTs) are a type of loadable bundle that low-level system routines recognize and load into the kernel environment. Although they are very similar to other loadable bundles, they cannot be loaded by applications or other non-kernel software. The KEXT bundles have an extension of `.kext`. See *Inside Mac OS X: Kernel Programming* for further information on KEXTs.

Although bundles can have any extension, there are some conventions. For example, applications typically use the `.app` extension. You can use other extensions for applications, as long as the application's `Info.plist` file has a CFBundlePackageType key with the value "`APPL`". The traditional extension for frameworks is `.framework`. Plug-ins and other loadable bundles can have any extension, but it should be an extension claimed by an application that knows how to load the bundle. The generic extension for loadable bundles is `.bundle`.

An Application's Main Bundle

With the exception of most command-line tools, every application has at least one bundle—its main bundle—which is the folder where its resources and executable files are located. Application bundles should have an extension of `.app` (for shipping applications), `.debug` (for applications with debug symbols), or `.profile` (for applications with profiling data). The Finder hides the `.app` extension from users.

Framework Bundles

Frameworks are bundles that package dynamic shared libraries, interface-definition files, images, and other resources that support the executable code along with the header files and documentation that describe the associated programming interfaces. As long as your applications are dynamically linked with frameworks, you should have little need to do anything explicitly with those frameworks thereafter; in a running application, the framework code is automatically loaded and linked, as needed. Frameworks should have an extension of `.framework`.

Loadable Bundles and Dynamic Linking

In addition to the main bundle and the bundles of linked-in frameworks, an application can be organized into any number of other bundles. Although these loadable bundles usually reside inside an application file package, they can be located anywhere in the file system (but typically are put in the `/Library/Extensions` directory of a file-system domain). An application can dynamically load the code and resources in a bundle when it needs them. For example, an application for managing PostScript printers may have a bundle containing PostScript code to be downloaded to printers.

The executable code in loadable bundles can be dynamically linked into an application while it runs. Using various code-loading programming interfaces, functions from loadable bundles can be looked up by name and called through function pointers. This newly linked code can then use a bundle identifier to obtain an instance for its bundle. Through this bundle instance, the code can locate and load additional resources packaged in the bundle.

Loadable bundles should have an extension. The conventional extension for loadable bundles is `.bundle` and, for Interface Builder palette bundles,`.palette`. Although the extension can be anything, it ideally should be an extension claimed by one or more applications that can load the bundle. These bundles are then associated with your application (by the Finder) and your application launches (if not started) and loads them when the user double-clicks them.

Localized Resources

If a bundle is to be used in more than one part of the world, its resources may need to be customized, or localized, for language, country, or cultural region. For example, an application may need to have separate Japanese, English, French, and Swedish versions of the character strings that label menu commands. An application may also need to accommodate regional language variation—British and American English, for example.

Bundles solve this problem by grouping resources together into directories named for their region and language with the extension `.lproj`. Region-specific resource directories should take their names from the ISO 3166 standard for country codes, and the ISO 639 standard for language codes (see http://www.iso.ch). You would place resources specific to the dialect of French spoken in France in a directory named `fr_FR.lproj`, whereas you would place resources specific to Canadian French in a directory named `fr_CA.lproj`. Localized resources that need not be region specific should be placed in directories named simply for the language, such as `English.lproj` or `Japanese.lproj`. These localized resource directories are then placed in a directory named `Resources` within the bundle's `Contents` directory. Nonlocalized (global) resources are kept in the top level of the `Resources` directory. See the section "Anatomy of a Bundle" (page 117) for an example of a complex bundle's file system layout.

The user determines which set of localized resources are actually used by the bundle at runtime. Bundle-related system routines rely on the language preferences set by the user in the Preferences application. Preferences lets users create an ordered list of available regions so that the most preferred region is first, the second most preferred region is next, and so on. When a bundle is asked for a resource file, it returns the file-system location of the resource that best matches the user's region preferences. See the section "Search Algorithm" (page 126) for details on the exact process Mac OS X uses to locate a bundle resource. See the section "Localizing File System Names" (page 176) for information on localizing bundle names. Also see the chapter "Internationalization" (page 209) for more detailed information about the naming of `.lproj` directories.

Localized Character Strings

One very common resource type is a strings file (which, by convention, has an extension of `.strings`). Strings files are used for character strings that must be localized. They are essentially dictionaries that map a string in the development language to the localized version of the string. The key is not required to be the development language version of the string, but this convention is usually used.

System routines know how to locate and load the strings file (like any other resource) and then look up the string you want all in one step. They also provide caching so multiple lookups from the same table do not require locating and loading the strings file again.

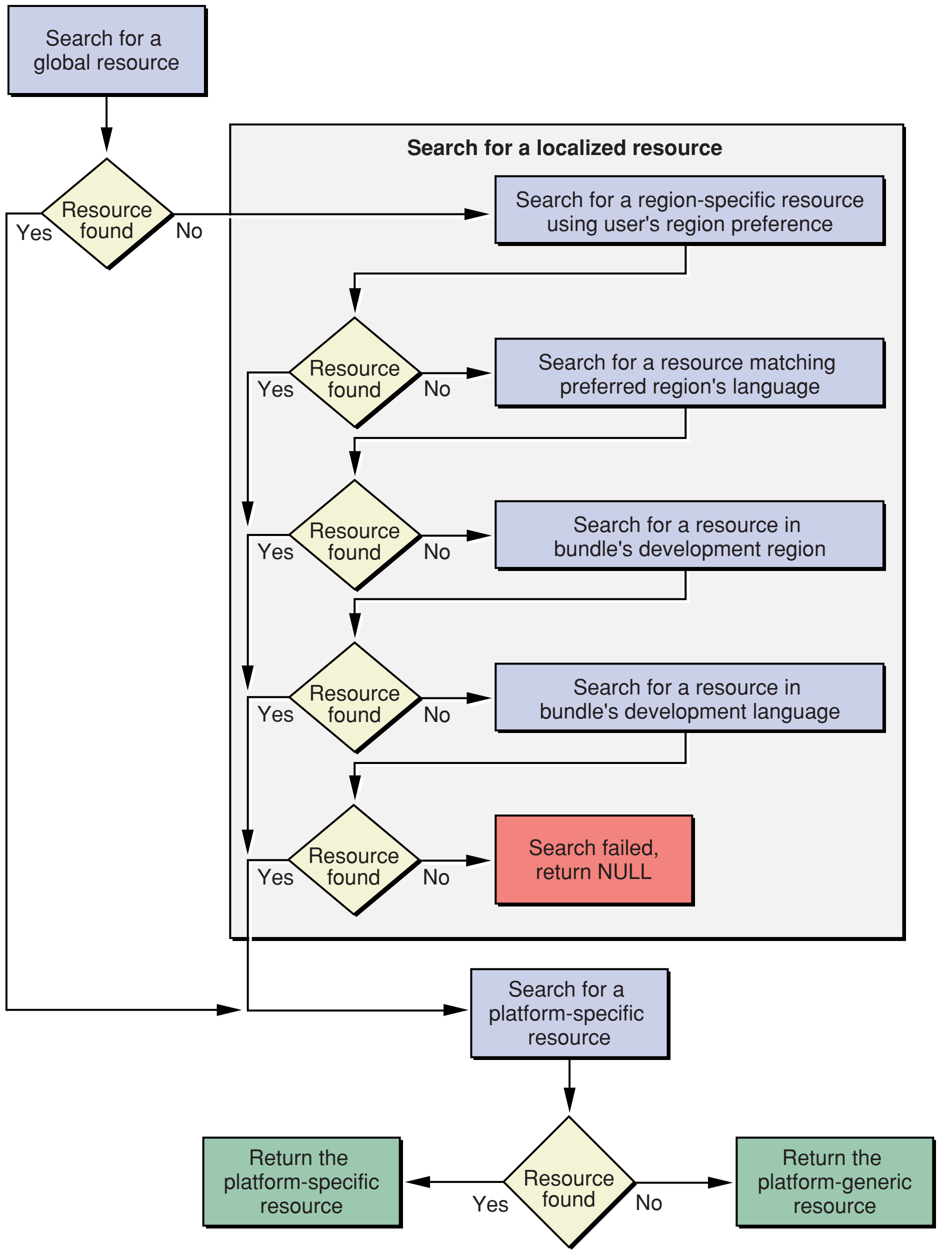
Because strings files are used so frequently, the Mac OS X development environments provide special macros and tools for working with them. For details, see "Localizing Strings" (page 217).

Search Algorithm

When you use a bundle-specific programming interface to locate a given resource, it performs a search to ensure that the right version of the resource is returned to you. Because resources can be global or localized as well as platform specific, the search may be complex. Various resource-finding APIs insulate you from potential changes to the bundle packaging scheme and handle a lot of tricky searching issues for you. You should always use these APIs instead of searching around inside the bundle yourself.

The Figure 5-2 details the steps a system routine uses to locate a resource.

Figure 5-2 Locating a resource in a bundle


Search for a global resource
Resource found
Yes
No
Search for a localized resource
Search for a region-specific resource using user's region preference
Resource found
Yes
No
Search for a resource matching preferred region's language
Resource found
Yes
No
Search for a resource in bundle's development region
Resource found
Yes
No
Search for a resource in bundle's development language
Resource found
Yes
No
Search failed, return NULL
Search for a platform-specific resource
Resource found
Yes
No
Return the platform-specific resource
Return the platform-generic resource

Notice that global resources take precedence over localized resources. In fact, there should *never* be both a global and localized version of a given resource. If there is a global version of a given resource, localized versions of that same resource will never be found. The reason for this precedence is performance. If the localizable resources were searched first, the bundle routines might search needlessly in several localized resource folders before discovering the global resource. Also notice that in order to find a platform-specific resource, the platform-generic version *must* exist. Again, the reason is performance. You should generally make one platform's version of the resource generic and provide platform-specific versions for any other platforms.

When a resource-locating routine finds a resource, it checks to see if a platform-specific version exists. Platform-specific resources are named using standard identifiers. The names you can use when making platform-specific resources are `macosclassic` (on Mac OS 9) and `macos` (on Mac OS X). You construct the name of a platform-specific resource by combining the platform-generic name with the platform identifier string. For example, if you have a resource named `Fish.jpg`, its name, when specific to Mac OS 9, would be `Fish-macosclassic.jpg`. When an application running on Mac OS 9 requests the resource `Fish.jpg`, the bundle routine also checks to see if `Fish-macosclassic.jpg` exists in that same folder. If it does, the routine returns the path to the platform-specific resource; if it does not, it returns the path to `Fish.jpg`. As was mentioned previously, for `Fish-macosclassic.jpg` to be found, a file named `Fish.jpg` must exist in the same folder (including language-specific resource directories).

Bundles and the Resource Manager

A bundle can contain any number of files containing Resource Manager–style resources in their data forks. These resource files—which, by convention, have an extension of `.rsrc` —are treated as bundle resources just as any other kind of file under the `Resources` directory. It is possible to use Core Foundation Bundle Services to get a CFURL type to such a file, convert that to an FSRef type, and then open it using the Resource Manager. There are, however, two special resource files that Bundle Services manages for you if you provide them. One is for nonlocalized resources, and it is called *executable name*`.rsrc`, where *executable name* is the name of your main executable. This file is stored with the other nonlocalized resources, in the `Resources` directory. The other file is for localized resources, and it is called

`Localized.rsrc`. This file is stored in the appropriate `.lproj` directory, one version for each language or region. Be sure that the resources are stored in the file’s data fork, not the resource fork.

When an application is launched, Bundle Services automatically attempts to open these files so that your application's resources are always available. For other bundles—frameworks and loadable bundles—you must do this yourself using the Bundle Services function provided specifically for this purpose.

If for some reason you are unable to convert your Carbon application to the bundle scheme, you can include the information property list (`Info.plist`) in your single-file application as a resource of type `'plst'`, id 0. See “CFM Executables” (page 229) for more information.

C H A P T E R 6

Application Packaging

A typical application in Mac OS X is not a single executable file but a package of files that includes one or more executable binaries. An application is a type of bundle—a directory in the system that contains, in a hierarchical organization, the application executable and the resources to support that code. An application is also a file package, a directory that the Finder presents to users as a file.

The design of application packages arises from the recognition that a running application is more than just the executable code that gets launched. Several advantages come with this internal organization, among them ease of installation and uninstallation, the inclusion of multiple localizations, support for multiple architectures and volume formats, and the capability for a single application to run, without modification, on Mac OS 9 and Mac OS X.

Although an application is structurally a bundle, some bundle components are found mainly, and sometimes only, in applications. Users tend to think of such things as help information, preferences, assistants, and plug-ins as application resources. Although, technically, nothing prevents these resources from belonging to, say, a loadable bundle, they are commonly associated with applications. This chapter focuses on how these resources are packaged in an application bundle. For a general description of bundles, see the chapter "Bundles" (page 115).

An Application Is a Bundle

An application in Mac OS X is packaged as a type of bundle. A bundle, to echo the definition in the chapter "Bundles" (page 115), is a directory containing executable code and the resources to support that code. Application bundles as well as loadable bundles (such as plug-ins) are file packages, directories that the Finder presents to users as a single file. The major distinguishing characteristic between the types of bundles—applications, frameworks, plug-ins, and other dynamically loadable packages of code and resources—is the nature of the executable. Application executables are generally self-sufficient binaries that users can launch from the Finder, usually by double-clicking. Applications may or may not contain secondary bundles, such as plug-ins, but they always contain their main bundle.

Bundles bring a number of benefits that are either specific to applications or that apply mostly to them:

- The same (Carbon) application bundle can run, without modification, on Mac OS 9 and Mac OS X.
- Applications can include different localizations. Applications can automatically display the set of localized resources (including the application name itself) that matches a user's language preference. Moreover, you can add a new localization to the application package, and it displays those resources (if they are for a preferred language) after the user relaunches the application.
- Client computers can run applications on a server.
- Customers can easily download applications from a website or obtain them through email.
- Applications are easy to install and uninstall; all the user must do is drag the application package onto a volume or, for uninstall, drag it to the Trash. (This feature does not preclude more complicated installations from taking place.)
- Because applications are file packages, users cannot "break" them by removing or changing essential parts of them. Users can, however, change the names of applications without affecting them.
- Applications can support multiple architectures as well as multiple volume formats.

What makes these features possible is the hierarchical internal organization of bundles. The different pieces of an application go in specific named locations within the application package. This standard internal organization of the pieces of an application enables related parts of the operating system—such as the Finder and the resource-finding and code-loading mechanisms of the system—to perform their functions. For example, executable files for multiple platforms (Mac OS 9 and Mac OS X) are put in separate subdirectories with standard names. The same goes for localized resources, plug-ins, and private and versioned frameworks.

The Finder, Sherlock, Navigation Services, and other Apple-provided applications and services that browse or examine the file system do not descend into application packages. The Finder responds to double-clicks on an application package by launching the application.

As they do with other bundles, Apple's development tools support the creation of application packages.

For additional general information about bundles, see the chapter "Bundles" (page 115). For further information about the Finder and its role in relation to applications, see the chapter "The Finder" (page 187).

Application Frameworks, Libraries, and Helpers

Applications sometimes have supplementary code modules—that is, code that isn't compiled into the application executable. This supplementary code may take the form of a framework, a shared library (CFM or otherwise), a plug-in, a helper application, or some other type of software.

There are various reasons for this compartmentalizing of application code. One is efficiency; for example, a software developer might have a suite of applications that all rely on the same framework or that make use of the same helper application, such as a custom document viewer. Another reason is performance; an application may decide to defer loading a module such as a plug-in until the user requests it. Or an application may be designed from the outset to be extensible.

The frameworks and shared libraries in the application package are those needed for the application to run, or at least to be complete. However, the application package does not include the Apple-supplied frameworks to which the application links. These are installed in the standard system location `/System/Library/Frameworks`. Installers should not delete frameworks in an application package or move them somewhere else.

The application bundle has two directories for the various types of supplementary code:

```
Frameworks/
PlugIns/
```

The remainder of this section explains the purposes and issues related to the first three of these directories. For a description of the `PlugIns` directory, see "Applications and Loadable Bundles" (page 135).

Private Frameworks

The `Frameworks` directory contains frameworks (or shared libraries) that are inextricably bound to the application. These frameworks are private to the application. Only the application itself uses the frameworks in this directory, and no other application does, including applications in the same "suite" or from the same developer. The dynamic shared libraries of these private frameworks are revision-locked and will not be superseded by any other, even newer, versions that may be available to the operating system.

An application always uses the code in `Frameworks`. If a framework or shared library is missing from `Frameworks`, the application cannot launch.

Listing 6-1 illustrates how a typical private framework might be stored in the application bundle.

Listing 6-1 Location of an application's private framework

```
FantasticApp.app/
    Contents/
        Info-macos.plist
        MacOS/
        Resources/
        Frameworks/
            GoodStuff.framework/
        PlugIns/
        SharedSupport/
```

Applications and Loadable Bundles

Loadable bundles contain code and programming resources that an application can dynamically load at runtime. The most common type of loadable bundle is a plug-in, but there can be others, such as Interface Builder palettes. Loadable bundles are somewhat different from frameworks and can have a slightly different relation with applications.

Loadable bundles are bundles just as much as application packages. They can thus contain all the things an application can, such as private frameworks and other supplementary code, including other plug-ins and other loadable bundles. (Frameworks, on the other hand, are "versioned" bundles with a different internal organization, among other differences. See the chapter "Frameworks" (page 139) for more information on frameworks.)

Plug-ins and other loadable modules are divided into three categories based on how essential they are to an application:

- those that an application requires to run
- those that are not essential to execution but that are considered part of an application because users generally want to use them (a tools palette, for example)
- those that meet neither of the above criteria but offer some additional functionality (often provided by third-party developers)

Plug-ins and other loadable bundles that meet the first two criteria should be packaged in the `PlugIns` directory of the application bundle. They should always be packaged with the application so they come along if the user moves the application to another location. If a loadable bundle is in the third category, the convention for users is to install it in the `Library/Application Support` directory of the logged-in user's home directory (local or remote). System administrators or expert users can install such a loadable bundle in the `Library/Application Support` directory of the system-local or network domains to make it more widely available.

Regardless of where a plug-in or loadable module is stored, it is the responsibility of the application to provide some human-interface mechanism enabling users to select them (as files, not directories). As an example, an application might display a dialog listing all plug-ins available for loading.

User Resources in Applications

An application can come packaged with a variety of resources. These resource can range from those that are closely tied to the application's executable, such as sound files and localized strings, to more "external" resources such as application help, preferences, and clip art. Resources are typically stored in the `Resources` directory of the application bundle.

However, application resources might not be stored in the application package for a variety of reasons. One reason for this separation is to make it possible for applications to run in a net-booted environment. Other reasons are to make the resources accessible in the file system and to separate resources provided by third-party contributors from those provided by the application's developer. The following sections give information about the preferred locations for several types of resources.

Application Help

On Mac OS X, the Help Viewer application displays help information for applications as well as more general help. (Help Viewer is part of the Apple Help product.) You should store application help files in the appropriate location in the application's `Resource` directory. You put the files in a help book, which is also the standard format for help in Mac OS 9. You internationalize help books by localizing their contents (text and images) and putting them in `.lproj` directories in the application bundle's `Resource` directory. The text files must be HTML 3.2-compliant and otherwise conform to the specification for Apple Help books. See *Inside Carbon: Providing User Assistance With Apple Help* for instructions on preparing and indexing help files.

The information property list (`Info.plist`) of an application must contain a key, CFBundleHelpBookFolder, whose value usually is a string that specifies the name of a help-book directory in the application's `Resources` directory. If the key CFBundleHelpBookName is also present, and the string value of this key is the `AppleTitle` tag of the book, Help Viewer automatically handles the display of the book when the user chooses the Help menu item. Note that the name of each help-book directory, regardless of localization, must be the same as the value for CFBundleHelpBookFolder—that is, directory names should not be localized. Your application also can control how help is opened and presented. The Apple Help APIs give applications several options related to help, such as opening a help book using the title, opening a help file using the full directory path to the file, and performing a search for a particular term or anchor. For more information, see *Inside Carbon: Providing User Assistance With Apple Help*.

Although Apple strongly recommends that you put help for an application in an application bundle, you can also put application help as well as more general help outside the application package. Such help should go in one of the standard locations for third-party help, including `/Library/Documentation/Help` and `Library/Documentation/Help` in a user's home directory. When the user launches the Help Center from the Finder, Help Viewer scans these locations and displays a link to the application help. If your application help is installed in one of the standard locations for help, you do not need to specify any special key-value pairs in the application's information property list for Help Viewer to handle it.

Application Preferences

Applications typically are installed with a default set of preferences that users can then change to suit their working habits. Part of any application's code is devoted to displaying the range of preference options, accepting user choices, and writing these choices to the preferences system (see "The Preferences System" (page 203) in the chapter ""Software Configuration" (page 197)").

Your application should never write user-preference data inside the application package. Preferences are stored in the `Library/Preferences` directory of the logged-in user's account (local or network) or in the same location in the machine-local or network domains. You should never write preferences data directly to these locations; instead use the APIs offered by Core Foundation Preference Services (CFPreferences) or, for Cocoa applications, NSUserDefaults. Part of the reason for the separation of user preferences from the application package is to make it possible for applications to run in a net-booted environment.

Document Resources

Applications that are document-centric—word processors, spreadsheets, drawing applications, to name a few—often include resources such as templates, clip art, tutorials, and assistants. These items can either be packaged in the application bundle or in a location external to the application package. The rule of thumb for deciding where such a resource goes is similar to that for plug-ins and other loadable bundles (see "Applications and Loadable Bundles" (page 135):

- If the resource is provided by the application developer, it should go in the `SharedSupport` directory of the application bundle.
- If the resource is from a third party, it should go in one of the standard directory locations for application resources, such as in the user's `Library/Application Support` folder.

As with loadable bundles, the application should provide some kind of resource browser that displays application resources (both internal and external to the package) and allows the user to select from them. The browser, however, should not divulge the inner structure of the application package.

Frameworks

A framework is a type of bundle that packages shared resources, such as a dynamic shared library, resource files, header files, and reference documentation. The consolidation of these resources brings with it a number of benefits. For example, it makes it easier for a dynamic shared library to locate its resources, and it makes installation and uninstallation easier on the user.

A framework bundle has an extension of `.framework`. Inside the bundle there can be multiple major versions of the framework. A network of symbolic links at the top level of the framework folder points to the most recent versions of library code and resources. The dynamic link editor writes the directory location in which to install a framework into the framework executable. When a program is launched, if the dynamic link editor cannot find a framework in this location, it looks in the standard directory locations for the framework. System and third-party frameworks are often installed in standard directory locations. Third-party frameworks can also be included in application packages that need those frameworks.

The executable code in a framework is a dynamic shared library. Multiple, concurrently running programs can share the code in this library without requiring their own copy. Unlike statically linked shared libraries, the binding of undefined symbols in a program linked with a dynamic shared library is delayed until the execution of the program. The dynamic link editor attempts to resolve undefined symbols at runtime when those symbols are referenced in the program. If a symbol in a library module is not referenced in a program, that module is not linked. The installation paths of dynamic shared libraries are written into all executables built with those libraries.

Frameworks can have major (or incompatible) versions and minor (or compatible) versions. The major versioning scheme provides for backward compatibility. Frameworks that are incompatible with programs linked with a previous version of the library are given a new major version. Those programs must link with an earlier version, which is kept inside the framework bundle. The minor versioning scheme provides for forward compatibility. A major version of a framework can incorporate a number of minor versions. A minor version denotes the framework compatibility of programs linked with later builds of the framework.

Note: Frameworks in the current release of Mac OS X are "versioned" bundles. Their internal directory structure lacks many of the features of the newer types of bundles, applications and loadable bundles. See "Anatomy of a Bundle" (page 117) in the chapter ""Bundles" (page 115)" for a description of new-style bundles.

The Framework as a Library Package

When libraries are installed in some computing environments, they are put in one location in the file system and resources related to that code are installed elsewhere. These related resources include header files as well as things such as images and localized strings. This scattering of code and resources can contribute to several problems:

- It complicates uninstallation of the library and its resources.
- It leads to a greater risk of mismatches between libraries and header files.
- It can make it more difficult for library code to locate resources.

Frameworks solve this problem by bundling a dynamic shared library with the resources used by the library or otherwise related to it. Indeed, "bundling" is an apt term because frameworks are bundles as much as applications and plug-ins are. However, frameworks differ in some significant ways from other types of bundles:

- Frameworks include a unique type of resource—header files. They can also contain as a resource anything else that is appropriate, such as private static libraries.
- The bundle bit is not set when a framework is built. As a result, the Finder does not treat the framework as a file package—a directory presented as a file—and thus developers can browse the packaged header files.
- Frameworks are versioned bundles, which are described in "The Internal Structure of Frameworks" (page 141).

Versioned bundles have an internal structure derived from Mac OS X Server (and prior) bundles. Apple will eventually convert frameworks to the new internal structure. Until then, Mac OS X will support both styles of bundles; the system routines for bundles can deal with both versioned bundles and the more recent type of bundles.

The Internal Structure of Frameworks

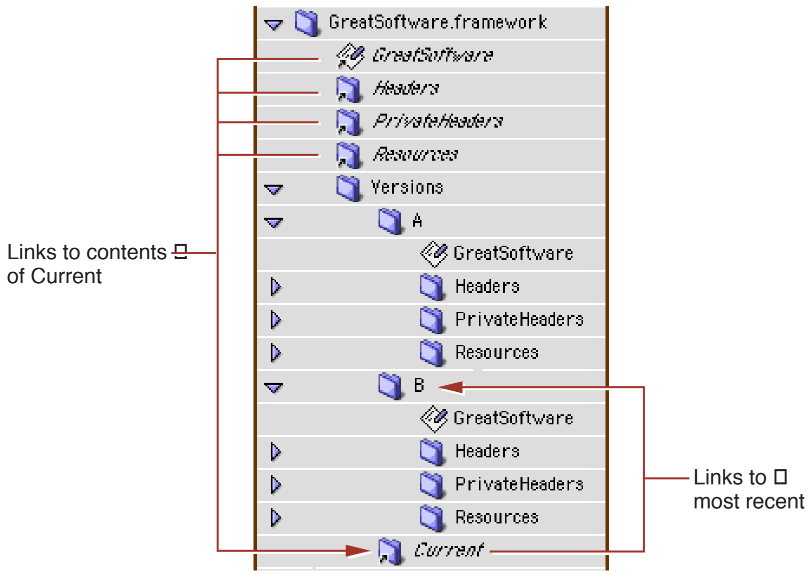
A framework in Mac OS X is a directory with an extension of `.framework`. When you open the directory, the first level of its contents looks something like this:

```
GreatSoftware.framework/
    GreatSoftware
    Headers/
    PrivateHeaders/
    Resources/
    Versions/
```

The `GreatSoftware` item is, in this example, the dynamic shared library. `Headers` and `PrivateHeaders` are subdirectories that store the framework's public and private header files. The framework's resources—items such as interface definition files, images, sounds, and localized strings—go in the `Resources` subdirectory.

The `Versions` subdirectory is the only one at this level that is a "real" directory. `GreatSoftware`, `Headers`, `PrivateHeaders`, and `Resources` are all symbolic links (similar to aliases) to the library and directories of the current major version of the framework. Figure 7-1 illustrates how this linking is done.

Figure 7-1 The directory structure of a framework



A framework directory can contain multiple major versions of dynamic shared libraries (along with their resources). A new major version of a framework is typically required when the dynamic shared library is not compatible with programs linked with prior versions of the library. Those applications will not run with the newest version but will run with an older one, so the older version is included in the framework bundle. Each version of the framework is contained in a subdirectory of `Versions` named, by convention, with letters of the alphabet. For more on major and minor framework versions, and on versioning in general, see "Framework Versioning" (page 146).

The contents of the `Resources` directory of frameworks is similar to that for new-style bundles. Localized resources are put in subdirectories of `Resources`. Each of these subdirectories has a name indicating a language (and possibly a region where that language is spoken) and an extension of `.lproj`. Resources specific to a region in which a language dialect is spoken should take their names from the ISO 3166 standard for country codes and the ISO 639 standard for language codes (with an underscore character separating the codes). For example, resources specific to Canadian French would go in resource directory `fr_CA.lproj`. But if you want one directory to hold all resources for all dialects of French, its name would be

`French.lproj`. The `.lproj` directories hold strings, images, sounds, and interface definitions localized to that language and locale. An important way in which frameworks differ from new-style bundles is that nonlocalized resources in frameworks do not go in a `Nonlocalized Resources` subdirectory of `Resources`; instead, they are put in the top level of the `Resources` directory.

Standard Locations for Frameworks

A system framework is a framework provided by Apple, such as the Application Kit or the QuickTime framework. The shared library code in system frameworks is intended for use by all applications on a system. System frameworks are installed in `/System/Library/Frameworks`. Third-party frameworks can go in a number of different file-system locations, depending on certain factors.

- If they are to be used only by a single user, they should be installed in the `Library/Frameworks` subdirectory of the user's home directory.
- If they are to be used by all users of a particular Mac OS X system, they should be installed in `/Library/Frameworks`.
- If they are to be used across a local area network, they should be installed in `/Network/Library/Frameworks`.

When you build an application or other executable, the compiler looks for imported frameworks in `/System/Library/Frameworks` as well as any other location specified to the compiler. The paths where required frameworks are expected to be installed are written into the executable itself, along with version information.

When an application is run, the dynamic link editor first tries to link with the frameworks whose installation paths are written into the executable. If it cannot find a framework in a specified location (perhaps it has been moved or deleted), it looks for frameworks in these standard fallback locations, in this order:

```
~/Library/Frameworks
/Library/Frameworks
/Network/Library/Frameworks
/System/Library/Frameworks
```

If the dynamic link editor cannot locate a required framework, it generates a link edit error and the application will not launch.

Dynamic Shared Libraries

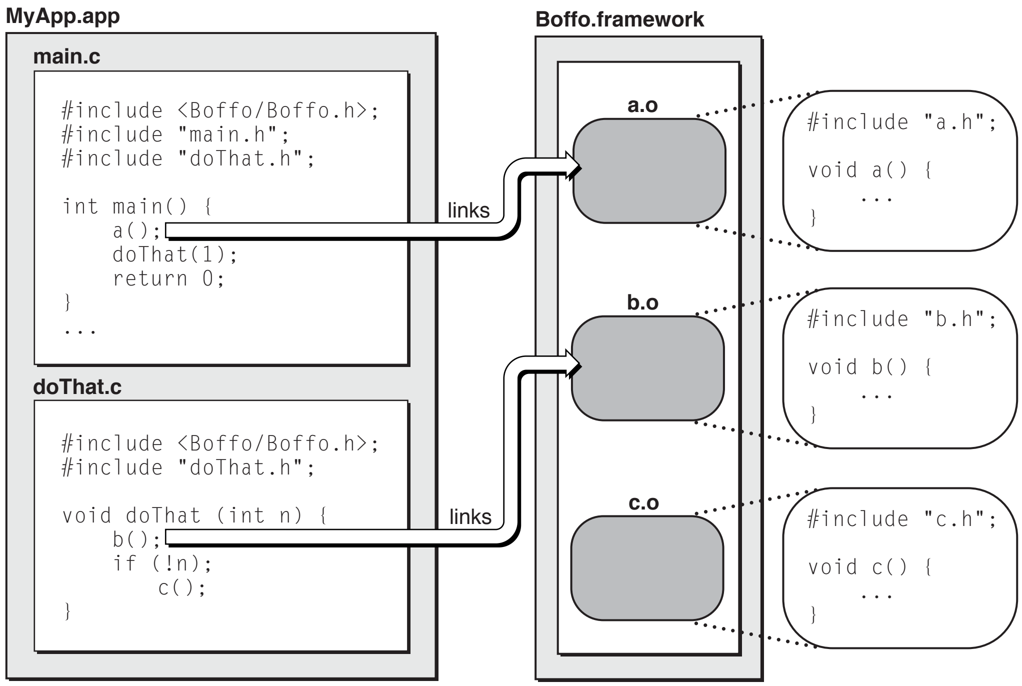
The executable code in a framework bundle is a dynamically linked shared library—or, simply, a dynamic shared library. This is a library whose code can be shared by multiple, concurrently running programs. Programs share exactly one physical copy of the library code and do not require their own copies of that code. Dynamic shared libraries bring several benefits. They enable more efficient use of memory and allow developers to fix bugs in library code and test those fixes without the need to rebuild programs that use those libraries.

Note: Although you can create dynamic shared libraries that reside outside a framework, this is an uncommon approach. Stand-alone dynamic shared libraries take, by convention, the extension `.dylib` and typically are installed in the standard file-system locations for libraries.

Dynamic shared libraries have characteristics that set them apart from statically linked shared libraries. With dynamic shared libraries, the binding of undefined symbols in a program is delayed until the execution of that program. In other words, the dynamic link editor not only attempts to resolve all undefined symbols at runtime, but attempts to do so only when those symbols are referenced during program execution. If an undefined symbol is not referenced, the binding is not needed for that particular execution of the program.

This dynamic behavior derives from the composition of dynamic shared libraries. The object-code modules from which a dynamic shared library is built retain their individual boundaries; that is, the code from the source modules is not merged into a single code base. When a program linked with a dynamic shared library is launched, the dynamic link editor automatically loads and links modules in the library, but it links them only as they are needed; in other words, a module is linked only if a symbol is referenced or a function is invoked that is defined in that module. If code in a module is not referenced or invoked, the module is not linked. Figure 7-2 illustrates this "lazy linking" behavior. In this example, module `a.o` is linked in the program's `main` routine when library function `a` is called; module `b.o` is linked when library function `b` in program function `doThat` is invoked; module `c.o` is never linked because its function is never called.

Figure 7-2 Lazy linking of dynamic shared library modules



```
#include <Boffo/Boffo.h>;
#include "main.h";
#include "doThat.h";

int main() {
    a();
    doThat(1);
    return 0;
}
...
```

```
#include <Boffo/Boffo.h>;
#include "doThat.h";

void doThat (int n) {
    b();
    if (!n);
        c();
}
```

```
#include "a.h";

void a() {
    ...
}
```

```
#include "b.h";

void b() {
    ...
}
```

```
#include "c.h";

void c() {
    ...
}
```

As a framework developer, you should design your dynamic shared library with this as-needed linking of separate modules in mind. Because the dynamic link editor always attempts to bind unresolved symbols within the same module before going on to other modules and other libraries, you should ensure that interdependent code is put in its own module. For example, custom allocation and deallocation routines should go in the same module. This technique prevents the wrong symbol definitions from being used. This problem can occur when definitions of a symbol exist in more than one dynamic shared library and those other symbol definitions override the correct one.

When you create a framework, you must ensure that each symbol is defined only once in a library. In addition, “common” symbols are not allowed in the library; you must use a single true definition and precede all other definitions with the `extern` keyword in C code.

When you build a program, linking it against a dynamic shared library, the installation path of the library is recorded in the program. For the system frameworks supplied by Apple, the path is absolute. For third-party frameworks, the path is relative to the application package that contains the framework. This capture of the library path improves launching performance for the program. Instead of having to search the file system, the dynamic link editor goes directly to the dynamic shared library and links it into the program. This means, obviously, that for a program to run, any required library must be installed where the recorded path indicates it can be found, or it must be installed in one of the standard fallback locations for frameworks and libraries.

Dynamic shared libraries can have dependencies on other dynamic shared libraries and these dependencies are recorded in the library executable. When the dynamic link editor links a program against the first dynamic shared library, it can obtain the paths of these dependent libraries and link against those as well. Thus the users of a dynamic shared library do not have to be aware of any dependencies when linking their programs against it.

Dynamic shared libraries can also be versioned, enabling backward compatibility and some degree of forward compatibility. See "Framework Versioning" (page 146) for more on this subject.

Framework Versioning

You can create different versions of frameworks based on the type of changes made to their dynamic shared libraries. There are two types of versions: major (or incompatible) and minor (or compatible) versions.

Major Versions

A major version of a framework, also known as an incompatible version, is incompatible with programs linked with a previous version of the framework's dynamic shared library. If any such program tries to run against the newer version of the framework, it will probably experience runtime errors.

Because all major versions of a framework are typically kept within the framework bundle, a program that is incompatible with the current version can still run against the version it is compatible with. The path of each major version of a framework encodes the version (see "The Internal Structure of Frameworks" (page 141)). For example, the letter "A" in the path below indicates the major version of a hypothetical framework:

```
/System/Library/Frameworks/Boffo.framework/Versions/A/Boffo
```

When the program is built, this path is recorded in the program executable itself. When the program is run, the dynamic link editor uses this path to find the compatible version of the framework's library. Thus the major versioning scheme enables backward compatibility of a framework by including all major versions and recording the major version for each executable to run against.

You should make a new major version of a framework when any of the following changes renders the dynamic shared library incompatible with programs linked with previous versions of the library:

- removing public API, such as a class, function, method, or structure
- renaming public API
- changing the data layout of a structure or adding to, changing, or reordering the instance variables of a class
- adding methods to a C++ class
- changing the programmatic interfaces of public API

An example of the last sort of change would be changing the order of parameters in a function.

The most recently built major version of a framework is typically made the "current" version. Unless you specify otherwise, each program you build is linked against the current version of a library; older programs that you rebuild are linked against the current version as well. When frameworks are built, the build system automatically generates a network of symbolic links that point to the current major version of a framework. See "The Internal Structure of Frameworks" (page 141) for details.

When you create a new major version of a framework, your integrated development environment takes care of most of the implementation details for you. All you need to do is specify the major-version designator. A popular convention for this designator is the letters of the alphabet, with each new version designator "incremented" from the previous one. However, you can use whatever convention is suitable for your needs, for example "2.0" or "Two".

You can also make major incompatible versions of stand-alone dynamic shared libraries (that is, libraries not contained within a framework bundle). The major version of this type of library is encoded in the filename itself, for example:

```
libMyLib.B.dylib
```

Then, assuming that this library is the most recent major version, the symbolic link `libMyLib.dylib` is created to point to it. This creates the current major version of the dynamic shared library.

Minor Versions

Within a major version of a framework there can be a series of minor, or compatible, versions. The minor versioning of a framework determines its compatibility with programs linked with later builds of the framework within the same major version. The minor versioning scheme thus helps to establish forward compatibility. If programming interfaces have been introduced to a recent version of a framework, programs that are built against this framework may not work with earlier minor versions of the framework. The program might have references to those new APIs and thus, if it is launched, it would probably crash with link-edit errors. Minor versioning gives framework developers control over how old a version of the framework can be used with an executable linked with a more recent version.

The relationship between two version numbers—the current version and the compatibility version—specifies a framework's minor-version status in relation to a particular program. The current version of a framework is a number that is incremented each time a framework is rebuilt after a compatible change is made to it (that is, a change not requiring a new major version).

The type of change introduced in a framework affects the value of the second minor version number, the compatibility version. If the change is merely a bug fix or an enhancement that does not affect any public API, the compatibility version is left unchanged from its current value. If, however, you have *added* classes, methods, functions, structures, or any other public API to the framework, the compatibility version number should be set to the same value as the current version number.

Important
The *addition* of instance variables to Objective-C or C++ classes or the addition of C++ methods constitutes a major incompatible change, not a minor compatible change.

When a framework is built or rebuilt, its current version number and its compatibility version number are recorded in the framework's dynamic shared library. When you build a program that links against this framework, these same numbers are encoded in the program executable, along with the path of the framework (which contains the major version designator). When you try to run the program, the dynamic link editor compares the program's compatibility version number and the framework's compatibility version number; if the program's compatibility version is greater than the framework's compatibility version, the program does not launch.

The minor versioning scheme applies as much to stand-alone dynamic shared libraries as to frameworks.

Versioning Summary and Guidelines

In Mac OS X there are two types of versions for frameworks and dynamic shared libraries:

- Major incompatible version—Designates a framework that is incompatible with programs linked with a previous version of the framework's dynamic shared library (backward compatibility).
- Minor compatible version—Designates a framework that is compatible with programs linked with later builds of the framework within the same major version (forward compatibility).

Table 7-1 summarizes the salient facts about each type of version.

Table 7-1 Summary of framework versioning

Type of version	When required	What happens
Major/incompatible (backward compatible)	API changes (such as renamed functions); deleted API; new or reordered instance variables; new C++ methods.	Major version designator changed; new designator is reflected in framework path. Path of dynamic shared library recorded in programs built with framework.
Minor/compatible (forward compatible)	New function, method, class, structure, and so forth.	Current (minor) version number incremented; compatibility version set to the same value as current (minor) version. Values recorded in programs built with this framework.
None	Bug fixes, enhancements not affecting programmatic interfaces	Current (minor) version incremented; compatibility version remains the same. Values are recorded in programs built with this framework.

The `otool` command-line program displays output that can give you an idea of how versioning information is recorded in a program executable. To use this program, change directories to any Mac OS X application and enter the following in a Terminal shell: `otool -L` *appName* where *appName* is the name of the application.

If you don't change the framework's major version number when you need to, programs linked with it will fail in unpredictable ways. If you change the major version number and you don't need to, you're cluttering up the system with unnecessary frameworks.

The main trick is to avoid having to change the version number in the first place. Here are some ways to do this:

- Pad classes and structures with reserved fields. Whenever you add an instance variable to a public class, you must change the major version number because subclasses depend on a superclass's size. However, you can pad a class and a structure by defining unused ("reserved") instance variables and fields. Then, if you need to add instance variables to the class, you can instead define a whole new class containing the storage you need and have your reserved instance variable point to it.

  Keep in mind that padding the instance variables of frequently instantiated classes or the fields of frequently allocated structures has a cost in memory.
- Don't publish API unless you want your users to use it. You can freely change private API because you can be sure no programs are using it. Declare any API in danger of changing in a private header.
- Don't delete things. If a method or function no longer has any useful work to perform, leave it in the API for compatibility purposes. Make sure it returns some reasonable value. Even if you add additional arguments to a method or function, leave the old form around if at all possible.
- Remember that if you add API rather than change or delete it, you don't have to change the major version number because the old API still exists. The exception to this rule is instance variables. (You do have to change the compatibility version number, however.)

When you do a "make clean" with your integrated development environment, you delete the entire framework bundle in the project directory, which means it deletes the old binaries in addition to the current binary. The subsequent build creates only the current version. You have no way of retrieving the earlier versions. If you must perform a make clean, you'll need to create multiple copies of the project: one that builds the current version and one for each of the previous versions.

Umbrella Frameworks

An umbrella framework is a public system framework that includes and links with constituent subframeworks and other public frameworks provided by Apple. A subframework is a public but restricted system framework that typically packages a specific Apple technology such as Open Transport or QuickDraw.

As the word "umbrella" implies, an umbrella framework encompasses all the technologies and APIs that define an application environment or a layer of system software. It provides a layer of abstraction between what outside developers link their programs with and what Apple engineering provides as implementation. The internal composition of subframeworks is an implementation detail subject to change. Apple has put mechanisms in place to discourage developers from directly including and linking with subframeworks.

Umbrella frameworks are not recommended for third-party developers. Apple instead recommends that external developers package their frameworks in applications. See the chapter "Application Packaging" (page 131) for more information.

This chapter describes the various kinds of private and public frameworks, defines umbrella frameworks and subframeworks, illustrates the internal structure of umbrella frameworks, and offers guidelines for linking with umbrella frameworks.

Kinds of Frameworks

The major application environments of Mac OS X as well as the layers of system software—the Application Services, Core Services, and kernel environment layers—are packaged as umbrella frameworks. The definition of this term depends on a few concepts that require several stages of explanation.

First, what is a framework? A framework is a hierarchically structured directory that holds a dynamic shared library along with supporting resources. These resources include header files, reference documentation, image files, and localized strings. The chapter "Frameworks" (page 139) describes frameworks in detail. A framework is also a type of bundle, but it differs in significant ways from other types of bundles, such as applications and plug-ins; see the chapter "Bundles" (page 115) for detailed information on bundles.

Second, a framework can be one of several types, or "flavors." To begin with, a frameworks is either private or public. Private frameworks are used only for internal development and their APIs are not exposed to customers. By convention, they go in the `PrivateFrameworks` folder of the system's, network's, or a user's `Library` directory; however, if frameworks are closely bound to an application, they typically go inside the application package (see "Application Packaging" (page 131)). Public frameworks are shipped to customers and their APIs are exposed through their header files. By convention, they are installed in the `Frameworks` directory in the appropriate `Library` location.

Third, the public frameworks that Apple ships with Mac OS X come in three varieties: the simple kind of public framework, the subframework, and the umbrella framework. These frameworks are installed on the installation hard disk in `/System/Library/Frameworks`. Public frameworks in this directory may be of the simple sort—that is, neither umbrella framework or subframework—only if they have been widely used in prior versions of the operating system, such as Mac OS X Server. The Cocoa application environment's Foundation and Application Kit frameworks fall into this category.

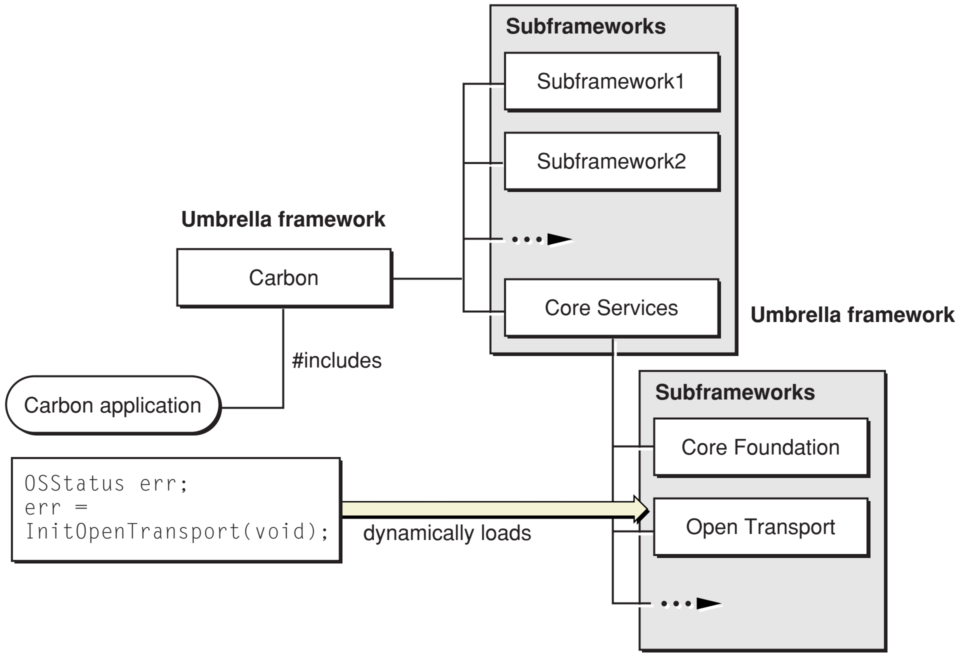
The Purpose of Umbrella Frameworks

An umbrella framework simply includes and links with constituent subframeworks and other public frameworks. An umbrella framework encompasses all the technologies and APIs that define an application environment or a layer of system software. It provides a layer of abstraction between what outside developers link their programs with and what Apple engineering provides as implementation.

A subframework is structurally a public framework that packages a specific Apple technology, such as Apple events or Quartz or Open Transport. However, a subframework is public with restrictions. Although the APIs of subframeworks are public, Apple has put mechanisms in place to prevent developers from linking directly with subframeworks (see "Restrictions on Subframework Linking" (page 160)). A subframework always resides in an umbrella framework installed in `/System/Library/Frameworks`, and within this umbrella framework, its header files are exposed (see "The Structure of an Umbrella Framework" (page 158)).

Some umbrella frameworks include other umbrella frameworks; this is particularly the case with the umbrella frameworks for the Carbon and Cocoa application environments. For example, both Carbon and Cocoa (directly or indirectly) import and link with the Core Services umbrella framework (`CoreServices.framework`). This umbrella framework, in turn, imports and links with subframeworks such as Core Foundation and Open Transport. Listing 8-1 (page 159) illustrates these relationships.

Figure 8-1 The relationship between an umbrella framework and its subframeworks



The exact composition of the subframeworks within an umbrella framework is an internal implementation detail subject to change. But by providing a level of indirection, umbrella frameworks insulate developers from these changes. Apple might restructure the subframeworks within an umbrella framework and might add, rename, or remove the header files within subframeworks. But these changes should not affect programs that link with the umbrella framework.

The value of an umbrella framework is that, by linking with it and only it, you can be assured that you have access to all the APIs necessary for programming in a particular application environment or layer of system software. Umbrella frameworks hide the complex cross-dependencies among the many different pieces of system software. Thus you do not need to know what set of frameworks and libraries you must import to accomplish a particular task. Umbrella frameworks also make faster builds possible because a precompiled header is included along with any umbrella header file or framework header file.

Linking and Including Guideline

For Mac OS X software developers the guideline for including header files and linking with system software is fairly straightforward: Include only the umbrella header file and link only with the umbrella framework appropriate to the program you are creating.

An umbrella header file includes the framework header files of its subframeworks. A framework header file (such as in a subframework) includes all the header files of the framework. You should never directly include the header files from subframeworks or link directly with them (and, in fact, you are prevented from doing so).

The general syntax of the command for including framework header files in Mac OS X is

```
#include <Framework/Header.h>
```

Where *Framework* is the name of the framework and *Header* is the name of a header file.

Note: For Objective-C projects, the `#import` directive may be used instead of `#include`; this directive is identical to `#include`, except that it makes sure that the same file is never included more than once.

To specify umbrella frameworks when developing software for Mac OS X, use the same `#include` syntax that is used for framework header files. In other words, to specify the Carbon umbrella framework, use the following command:

```
#include <Carbon/Carbon.h>
```

However, Apple provides an interim solution for Carbon developers porting their source code from Mac OS 9 to Mac OS X or otherwise writing code to be built on both operating systems. This “flat header” alternative allows them to continue using their present `#include` commands. In `/Developer/Headers/FlatCarbon` are stub files for all public Mac OS 9 header files. These stub files redirect the compiler to the appropriate umbrella header file or contain warnings if the API is not valid on Mac OS X. To make use of the stub files, you must use the compiler's `-I` flag (that is capital "I", not lowercase "l") to include the files in the `FlatCarbon` folder:

```
-I/Developer/Headers/FlatCarbon
```

Make sure that you include both `MacWindows.h` and `MacTypes.h`.

Once you are only compiling code for Mac OS X, you should start using the native syntax for including umbrella frameworks. (As a side effect of doing this, build time will decrease.) You can also conditionalize your `#include` commands so that they include umbrella frameworks directly (for example, `#include <Carbon/Carbon.h>`) when you are building on Mac OS X and include flat headers (for example, `#include <Dialogs.h>`) when you are building on Mac OS 9. This conditional approach obviates the need for the `-I` flag.

The book *Inside Carbon: Carbon Porting Guide* contains a more detailed discussion of the flat-header `#include` technique. Also see "The Structure of an Umbrella Framework" (page 158) for more information about umbrella header files.

Do not worry about bloating your program's memory footprint by linking it with an umbrella framework and including its (potentially) dozens of header files. Because the executable code of a framework is a dynamic shared library, a subframework's code is loaded into memory only when one of its functions or methods is first called. If your program does not use a subframework, it is not loaded. See "Dynamic Shared Libraries" (page 144) in the chapter ""Frameworks" (page 139)" for more on this subject.

The Structure of an Umbrella Framework

Two things determine the structure of an umbrella framework. The first is the manner in which it includes header files. The second is how it, as a bundle directory, organizes its subframeworks.

The `#include` examples in the previous section suggests how umbrella header files and framework header files are used to accomplish the level of abstraction afforded by umbrella frameworks. To reiterate, the general syntax of the `#include` command for including framework header files and umbrella header files is

```
#include <Framework/Header.h>
```

In this convention, the framework and the umbrella header file have the same name.

An umbrella header file includes the framework header files of its subframeworks. For example, the umbrella header for the Core Services umbrella framework, `CoreServices.h`, has contents similar to the following:

```
#include <CoreFoundation/CoreFoundation.h>
#include <OT/OT.h>
#include ...
```

The framework header file includes all the header files defining the public interface of a particular technology. `CoreFoundation.h`, for example, is the framework header for the Core Foundation subframework (`CoreFoundation.framework`). Its contents are similar to the following:

```
#include <CoreFoundation/CFBase.h>
#include <CoreFoundation/CFArray.h>
#include <CoreFoundation/CFBag.h>
#include ...
```

Physically, umbrella frameworks contain their subframeworks using a structure constructed from subdirectories and symbolic links (a mechanism similar to aliases). Listing 8-1 depicts a hypothetical framework. (Symbolic links in this example are items followed by an "at" sign (@); they include the referenced path.)

Listing 8-1 Structure of an umbrella framework

```
Umbrella.framework/
    Headers@ -> Versions/Current/Headers/
    PrivateHeaders@ - > Versions/Current/PrivateHeaders/
    Resources@ -> Versions/Current/Resources/
    Umbrella@ -> Versions/Current/Umbrella
    Versions/
    Frameworks/
        SubFW1.framework/
            SubFW1@ -> Versions/Current/SubFW1
            Headers@ -> Versions/Current/Headers/
            PrivateHeaders@ -> /Versions/Current/PrivateHeaders/
            Resources@ -> Versions/Current/Resources/
        Versions/
    SubFW2.framework/
        SubFW2@ -> Versions/Current/SubFW2
        Headers@ -> Versions/Current/Headers/
        PrivateHeaders@ -> /Versions/Current/PrivateHeaders/
        Resources@ -> Versions/Current/Resources/
        Versions/
```

Each subframework of the umbrella framework goes in the `Frameworks` directory. The `Headers` directory referenced by the umbrella framework's symbolic link contains the umbrella header file (`Umbrella.h` in the above example). The umbrella header file includes a `#pragma` command that tells the compiler where the subframeworks are located.

There are a couple of things to note about the structure of frameworks in general:

- The `PrivateHeaders` directory contains header files used in internal development and is not shipped to customers.
- Aside from the umbrella framework's `Frameworks` directory, the `Versions` subdirectory of a framework is the only "real" one—that is, the only directory at that level that isn't a symbolic link. It contains the major versions of a framework. The `Current` directory is a symbolic link that typically points to the most recent version. For more on the general structure of frameworks, see "The Framework as a Library Package" (page 140) in the chapter ""Frameworks" (page 139)."

Restrictions on Subframework Linking

Mac OS X includes two mechanisms for ensuring that developers link only with umbrella frameworks. One mechanism is triggered when you attempt to include subframework header files. The other mechanism prevents linking with subframeworks.

If, as an external developer, you try to link with a subframework, the linker causes the link to fail and displays a message explaining why. For example, if you try to link directly with the Open Transport framework (`OT.framework`), the link fails and the linker prints the following message: "OT.framework is a subframework. Link against the umbrella framework CoreServices.framework instead."

If you try to include a header file that is in a subframework, you get a compile-time error message. The umbrella header files and the subframework header files contain preprocessor variables and checks to guard against the inclusion of subframework header files. If you compile your project with an improper `#include` statement, the compiler generates an error message.

C H A P T E R 9

The File System

From an architectural perspective, Mac OS X implements multiple file systems, most importantly Mac OS Extended (HFS+), Mac OS Standard (HFS), UFS, ISO 9660, NFS, and AFP. But from a user perspective, the file systems are monolithic; when users copy, move, or drag files and folders, there is (or seems to be) one file system.

This chapter looks at file systems from both perspectives and discusses topics that are of interest to software developers. First it describes the standard directory layout in Mac OS X—where things like applications, documents, frameworks, and resources go in a multiuser, networked computing environment. Then it describes differences and issues of interoperability between the various file systems, particularly the dominant ones: HFS+ and UFS. It also explains the implementation of HFS resource forks and the policies related to this implementation.

How the File System Is Organized

In Mac OS X almost every file in the file system has its proper place—a standard directory location for files of that type. For users, this doesn't mean they *must* put their applications and application resources in the recommended locations. Applications, after all, are packaged so they can be self-sufficient wherever they're installed. But if users do not put certain things where system software expects them, they lose some advantages of Mac OS X. For example, the Finder first populates an application database by looking in the standard locations for applications ("Collecting Application Information" (page 190)). As a result, a document belonging to an application that is not in one of those locations might not immediately open when double-clicked.

Before exploring the rationale behind the file-system organization, consider what the Finder displays at the top level of the file system. Listing 9-1 illustrates a hypothetical installation.

Listing 9-1 The top level of a sample Mac OS X file system

```
/Mac OS X/
/Network/
/OtherVolumes/
```

The layout of a file system is often represented as a hierarchical tree structure that begins at a "root." At the root of a typical Mac OS X file system (root indicated by an initial `/`) are the following items:

■ `/Mac OS X/`—The volume from which the operating system boots and on which system software and resources are installed. This volume is typically a hard disk formatted to be a Mac OS Extended (HFS+) volume (although it can be a UFS volume). The name "Mac OS X" is the default volume name, which users can change.

■ `/Network/`—The root of the local area network, as mounted on the user's system. The `/Network/` directory (whose icon is a globe) always appears whether the user is connected to a network or not.

■ `/OtherVolumes/`—Represents one or more externally connected devices or internal devices that are not the boot volume. This includes items such as Zip drives, CD-ROM drives, digital cameras, and mounted network servers as well as hard disks and their partitions. (The name "OtherVolumes" is only representative; the actual name of each connected volume will be different.)

  All non-boot volumes appear as they are mounted and disappear when they are unmounted. An exception to this is the user's iDisk volume, which appears even when it is unmounted.

The physical organization of volumes is somewhat different than what the Finder presents to the user. If you were to look at the directory structure using the Terminal application, you would see that the boot volume is mounted at the root level (`/`) and non-boot volumes are located in `/Volumes/`. The Finder provides this abstraction to provide a more traditional Mac OS interface on top of the underlying UNIX system.

Also at the root level, but hidden from users by the Finder, are the standard BSD directories such as `/usr`, `/bin`, and `/etc`.

File-System Domains

On a multi-user system, controlling access to system resources is important for maintaining the stability of the system. Mac OS X defines several file system domains, each of which provides storage for resources in an established set of directories. Access to resources in each domain is determined by the permissions for the current user.

There are four file system domains, each of which is described in the following list:

- **User**. The user domain contains resources specific to the user who is logged into the system. This domain is defined by the user's home directory, which can either be on the boot volume (`/Mac OS X/`) or on the network. The user has complete control of what goes into this domain.
- **Local**. The local domain contains resources such as applications and documents that are shared among all users of a particular system, but which are not needed for that system to run. The Local domain does not correspond to a single physical directory, but instead consists of several directories on the local boot (and root) volume. Users with system administrator privileges can add, remove, and modify items in this domain.
- **Network**. The network domain contains resources such as applications and documents that are shared among all users of a local area network. Items in this domain are typically located on network file servers and are under the control of a network administrator.
- **System**. The system domain contains the system software installed by Apple. The resources in the system domain are required by the system to run. Items in this domain are located on the local boot (and root) volume. Users cannot add, remove, or alter items in this domain.

The domain for a given resource determines its applicability or accessibility to the users of the system. For example, a font installed in the user's home directory is only available to that user. If an administrator were to install the same font in the network domain, all network users would have access it.

Within each domain Mac OS X provides a set of initial directories for organizing the contained resources. Mac OS X uses identical directory names across domains to store the same types of resources. This consistency simplifies the process of finding resources both for the user and for the system methods that use those resources. When the system needs to find a resource, it searches the domains sequentially until it finds the resource. Searches start in the User domain and proceed through the Local, Network, and System domains in that order.

Your code should never assume the path to a resource within a file-system domain, as those paths could change in the future. Apple provides public APIs for accessing standard file system paths. You should always use these APIs to locate system resources. See "Searching Within the File-System Domains" (page 175) for more on searching for items within the domains.

The following sections describe each of the file system domains in more detail, including some of the standard directories available in that domain.

The User Domain

The user domain contains resources that are specific to a single user. The user domain is represented by the home directory of the current (logged-in) user. Each user of a Mac OS X computer must have an account on that computer or on the local area network to which the computer is connected. Each user account comes with an assigned area of space in the file system, called the user's home directory. This directory is where the user's programs, resources, and documents reside. The name of each user's home directory is based on the user's short login name, which must be unique.

The user domain makes a customized working environment possible for each user. When a user logs in, the Finder restores the user's working environment and settings to their previous state using the preferences in the user domain. Similarly, programs and other system software use information in the user domain to restore application preferences, network settings, email settings, font sets, ColorSync profiles, and other settings.

The location of the user's home directory—user domain—is dependent on the user account. If the user account is local to the computer, the user's home directory is in the `Users` directory on the boot volume. If the user account is a network account, the home directory is on a network server. Regardless of the physical location of the home directory, Mac OS X uses the UNIX convention of a ~ (tilde) character in some situations to indicate a user's home directory. The tilde character can be used in combination with other directory or user names to specify specific user directories. Table 9-1 illustrates this concept.

Table 9-1 Uses of tilde to indicate locations in home directories

`~`	Top level of current user's home directory
`~/Library/Fonts`	Where fonts are stored in current user's home directory
`~Steve`	Top level of user Steve's home directory

The home directory for each new user comes with some default directories and resources in place. The directories in a user's home directory mirror those found in iDisk accounts. (For more information on iDisk, see the iTools section of http://www.apple.com.) The default directories in a user's home directory are the same regardless of where the home directory is created. Table 9-2 (page 167) lists the standard subdirectories of a user's home directory:

Table 9-2 Default home directory contents

User Directory	Description
`Desktop`	Contains the items the Finder displays on the desktop for the logged-in user.
`Documents`	Contains the user's personal documents.
`Library`	Contains application settings, preferences, and other system resources that are specific to the user. See "The Library Directory" (page 170).
`Movies`	Contains digital movies in QuickTime and other formats.
`Music`	Contains digital music files (`.aiff`, `.mp3`, and other formats).

Table 9-2 Default home directory contents (continued)

User Directory	Description
`Pictures`	Contains image files in a variety of formats.
`Public`	Contains items the user wishes to share with other users. By default, this directory is accessible to other users.
`Sites`	Contains web pages for the user's personal web site. Web Sharing must be enabled before these pages are accessible to other users.

When a user account is created, an `Applications` directory is not automatically added to the home directory. However, users can create an `Applications` directory and put their own applications in it. The system automatically searches for applications in this location.

The system protects the files and directories in the user's home directory from outside interference by a set of default permissions, which the user may change at any time. Any new folders created by the user inherit the privileges of the parent directory.

In addition to the individual home directories, the `Users` directory contains a `Shared` subdirectory. This directory is accessible to any user of the local computer system. Any user can write documents to, retrieve documents from, and read documents in this directory. Although this directory is not really associated with the user domain, it provides a convenient means for users to exchange documents and other files.

The Local Domain

The local domain contains resources that are available on the local computer but are not required by the system to run. Resources in the local domain typically include applications, utilities, custom fonts, custom startup items, and global application settings. The `Applications` and `Library` directories on the root volume contain the resources for the local domain. These resources are available to the current user of a computer system but are not available to users on other networked computers.

Administrators of a computer can install resources into the local domain if they want those resources to be shared by all users of the system. Apple ships its applications in the `/Applications` and `/Applications/Utilities` directories. Third party applications and utilities should also be placed in these directories. Other system resources, such as fonts, ColorSync profiles, preferences, and plug-ins should be placed in the appropriate subdirectory of the `Library` directory. For more on the `Library` directory, see "The Library Directory" (page 170).

The Network Domain

The network domain contains the resources available to all users of a local area network. Network users can access applications, documents and other resources through this domain, including AppleShare and Web servers. The exact composition of the network domain depends on institutional or corporate policy. Implementation of the network domain is the responsibility of the network administrator.

Table 9-3 lists the standard directories available in the Network domain, along with a description of the directory contents.

Table 9-3 Network directories

Location	Description
`/Network/Applications`	Contains applications that can be run by all users on the local area network.
`/Network/Library`	Contains resources—such as plug-ins, sound files, documentation, frameworks, colors, and fonts—available to all users of a local area network. For more on the `Library` directory, see "The Library Directory" (page 170).
`/Network/Servers`	Contains the mount points for NFS file servers that make up the local area network.
`/Network/Users/`	Contains the home folders for all local-area network users. This is the default location for home folders. User home folders may also be stored on other servers.

The System Domain

The system domain contains the resources required by Mac OS X to run. All resources in the system domain are located in the `/System` directory on the root volume. These resources are provided by Apple and only the root user can modify the contents of this directory. Administrative users and applications cannot install resources in the system domain or modify its contents directly.

By default, the `/System` directory contains only a `Library` subdirectory. This subdirectory contains many of the same types of resources as other `Library` directories in the system. However, in the system domain, this directory also contains the core services, frameworks, and applications that make up Mac OS X. For more information on the `Library` directory, see "The Library Directory" (page 170).

Although the Classic compatibility environment contains system-related resources, it is not considered a part of the system domain. For more information about the Classic environment, see "Directories of the Classic Environment" (page 174).

The Library Directory

The `Library` directory is a special directory used to store application-specific and system-specific resources. Each file system domain has its own copy of the `Library` directory, with access levels to match the domain type. Although an application can use this directory to store internal data or temporary files, it is not intended for storage of the application bundle itself or for user data files. Application bundles belong in an appropriate `/Applications` directory, while user data belongs in the user's home directory.

The `Library` directory contains many standard subdirectories. System routines expect many of the standard subdirectories to exist, so it is never a good idea to delete subdirectories of `Library`. However, applications can create new subdirectories as needed to store application-specific data.

Table 9-4 lists some of the directories that can appear in a `Library` directory. This list is not complete, but it lists some of the most relevant directories for developers. Directories that do not appear in all domains are noted appropriately.

Table 9-4 Subdirectories of the `/Library` directory

Subdirectory	Directory contents
`Application Support`	Third-party plug-ins, helper applications, templates, and other resources for a specific application. By convention, these items should be put in a subdirectory named after the application. For example, third-party resources for the application MyApp would go in `Application Support/MyApp/`. Note that resources created by the developer of an application should go in the application package itself. See "Application Packaging" (page 131) for more information.
`Assistants`	Programs that assist users in configuration or other tasks.
`Audio`	Audio plug-ins and device drivers.
`ColorPickers`	Resources for picking colors according to a certain model, such as the HLS (Hue Angle, Saturation, Lightness) picker or RGB picker.
`ColorSync`	ColorSync profiles and scripts.
`Components`	System-wide components and extensions.
`Documentation`	Documentation files and Apple Help packages (in subdirectory `Help`) intended for the users and administrators of the computer. In the local domain, this directory contains the help packages shipped by Apple (excluding developer documentation).
`Extensions`	Device drivers and other kernel extensions (system domain only).
`Favorites`	Aliases to frequently accessed folders, files, or websites (user domain only)
`Fonts`	Font files for both display and printing
`Frameworks`	Frameworks and shared libraries

Table 9-4 Subdirectories of the `/Library` directory (continued)

Subdirectory	Directory contents
`Internet Plug-ins`	Plug-ins, libraries, and filters for the Internet
`Keyboards`	Keyboard definitions
`Mail`	Contains the user's mailboxes (user domain only)
`Preferences`	User preferences. See "The Preferences System" (page 203) in the chapter ""Software Configuration" (page 197)."
`Printers`	Print drivers (by vendor) and PPD plug-ins
`QuickTime`	QuickTime components and extensions
`Scripting Additions`	Scripts and scripting resources that extend the capabilities of AppleScript
`Sherlock Plug-ins`	Plug-ins for extending the capabilities of Sherlock
`Sounds`	System alert sounds
`StartupItems`	System and third-party scripts and programs to be run at boot time. See "Startup Items" (page 93) in the chapter ""Booting and Logging In" (page 89)" for more information
`Web Server`	Web server content. This directory contains the CGI scripts and web pages to be served.

The Developer Directory

You can install the applications, tools, documentation, and other resources for developing Mac OS X software as an optional package. When you install the developer tools, the installer places all of the software components in the `Developer` directory, located in the boot volume (`/Mac OS X`).

Table 9-5 shows the contents of the `Developer` directory.

Table 9-5 Developer subdirectory contents

Directory	Contents
`Applications`	Applications used to manage and build software projects (Project Builder), to create user interfaces (Interface Builder), and to performance-tune programs.
`Documentation`	Developer documentation.
`Examples`	Example projects organized by general type (Carbon, Java, and so on).
`Headers`	Special header files, such as the stub "flat" Carbon headers.
`Java`	Files needed for Java bridging in the Cocoa application environment.
`Makefiles`	Makefiles and jamfiles for building and converting projects.
`Palettes`	Apple-supplied Interface Builder palettes.
`PBBundles`	Loadable bundles used by Project Builder.
`ProjectBuilder Extras`	Project Builder templates and plug-ins.
`ProjectTypes`	Definitions of project types used by Project Builder.
`Tools`	Command-line development tools, including those for creating and manipulating HFS resource forks.

Project Builder defines a set of makefile variables that your projects should use when specifying locations within file-system domains. You should use these variables instead of hard-coding directory paths because those locations are subject to change. For a complete list of build settings (including the makefile variables), see the Project Builder Help.

Directories of the Classic Environment

The Classic environment contains several directories used to support Classic applications. The directories of the Classic environment are the directories of a Mac OS 9 installation. Mac OS X requires an installation of Mac OS 9.1 (or later) for the Classic environment. If a system has an earlier version of Mac OS 9 installed, the user must install a newer version to support Mac OS X.

A system may have multiple versions of Mac OS 9 installed on different partitions. If this is the case, the Classic pane of System Preferences lets the user choose which version of Mac OS 9 to use for the Classic environment. The first time the user launches Classic, the system adds some required files to the System Folder of the selected Mac OS 9 volume. The user can switch Classic environments at any time using the Classic pane of System Preferences. The user can also change the startup disk to boot directly into Mac OS 9 instead of Mac OS X using the Startup Disk System Preference.

When you install Mac OS 9.1 (or later) on a volume, the installer creates several directories to store the system files. Table 9-6 lists the directories created by the installer along with a description of the contents. If you already have a version of Mac OS X or Mac OS 9.1 (or later) installed, the Mac OS 9 installer may not create all of these directories.

Table 9-6 Directories created by the Mac OS 9.1 (or later) installer

Directory	Description
`Applications (Mac OS 9)`	Contains the Mac OS 9 (Classic) applications and utilities.
`Documents`	Contains application-specific information. This directory should be used only by Classic applications. Mac OS X applications should store preferences and other application files in the appropriate `/Library` directory. Users should store their documents in their home directory.
`System Folder`	Contains the Classic environment system files.

When you install Mac OS X on a system with Mac OS 9 already installed, the installer performs some additional tasks to support the Classic environment. In particular, the Mac OS X installer creates an alias to the Mac OS 9 desktop folder and puts it on the desktop of the administrative user who ran the installer. This alias contains links to any files that were on the Mac OS 9 desktop prior to the Mac OS X installation.

Searching Within the File-System Domains

Mac OS X includes two public programmatic interfaces you can use to search for resources, plug-ins, and other items within specific directory locations of specific (or all) domains. One of these APIs—the `FindFolder` function of the Folder Manager—is for Carbon programs. The other API—the functions and constants defined in `NSPathUtilities.h` in the System framework—is for any other type of program.

Both APIs help you search through all file-system domains for a particular item. By convention, searches typically begin with the most specific domain and end with the most general. This domain order is as follows:

1. User
2. Local
3. Network
4. System

Most system software follows this order when it searches for items through all file-system domains. However, you may search in any domain order that is appropriate to your application's needs.

Hiding Filename Extensions

In Mac OS X, the type of a file can be identified using two separate techniques: file types and filename extensions. File types are common to Mac OS 9 and are stored as file metadata. Filename extensions are commonly used on Windows and UNIX operating systems are supported by Mac OS X to provide maximum compatibility with other operating systems. However, in order to preserve the Macintosh user experience, Mac OS X provides a way to hide file extensions on a per-file basis.

Each file in the file system now has a special flag identifying whether the file's extension is hidden or shown. This setting affects only the way the file is displayed; it does not physically change the name of the file in the file system. Users can change this setting for individual files from the Info panel of the file. Users can also hide all filename extensions by modifying the Finder preferences.

Applications that display filenames as part of their user interface must use special routines for getting the display name of a file. The Launch Services methods `LSCopyDisplayNameForRef` and `LSCopyDisplayNameForURL` take the file extension flag into account when returning a file name, as does the NSFileManager method `displayNameAtPath`. Your application should use these methods only when displaying the file in your user interface. Any other internal file manipulations made by your application should always use the full file name.

Applications also need to be aware of filename extension hiding preferences when opening and saving files, including the following:

- Save dialogs should allow users to control whether to hide filename extensions.
- Applications should preserve the existing show/hide setting and filename extension when opening or saving a document.
- Applications should add an appropriate extension when saving a new file or when saving an existing file using the Save As command.
- Applications should not append an extension or change the show/hide setting of a file that does not have an extension.

Both Carbon and Cocoa enable filename extension hiding in the corresponding Save dialogs. In Carbon, you can set the `kNavPreserveSaveFileExtension` dialog option for the Navigation Services save dialog. For Cocoa, you can use the `setCanSelectHiddenExtension:` method of NSSavePanel to enable this feature. For more information, see the Carbon and Cocoa references.

Localizing File System Names

Mac OS X supports multiple simultaneous localizations for select file system elements. Localization support for file system elements is intended to provide users with a more complete localization than they might previously have had. Previously, when users changed the language settings of a system, menu bars and UI elements would change while the file system elements, such as system folder names, would retain their default names. Now, select file system names change to match the currently selected language. The result allows users to navigate more of the file system hierarchy in their native language.

Mac OS X applies localization support primarily to system directory and bundle names, including application names. Directory names such as System, Applications, and Library among others now appear in the currently selected language of the logged in user. Developers can also take advantage of localization support for their own application and directory names. For information on localizing application names, see "Localizing Bundle Names" (page 177). For information on localizing directory names, see "Localizing Directory Names" (page 178). File extensions are never localized.

Important
Mac OS X does not support localized bundle and directory names in the Darwin and Classic environments, nor does Mac OS X support the localization of flat files.

Displaying Localized Path Names

Developers need to be aware of localized file system names in their applications and display them appropriately. Localized paths are intended only for display to the user through your application user interface. You should never attempt to use localized file paths to access files directly, nor should you store localized paths in preference files or internal caches. Localizing a path should always be the last step prior to displaying that path to the user.

Mac OS X provides several functions for obtaining the localized name of a path. Your application should always use one of these functions to convert a path to its localized name immediately prior to display. In addition to localizing appropriate directory and bundle names, these methods also hide file extensions when called for by the current file and Finder settings. The Launch Services methods `LSCopyDisplayNameForRef` and `LSCopyDisplayNameForURL` convert paths to their localized formats, as does the NSFileManager method `displayNameAtPath`.

Localizing Bundle Names

Bundles and applications support localized file names through the existing bundle localization mechanism. The Resources folder of a bundle can contain multiple `.lproj` subdirectories, each containing the localized resources for one language. One of the files you can put in these language subdirectories is a `InfoPlist.strings` file, which stores localized values for special keys. To specify a localized name for your bundle, include the CFBundleDisplayName key in this file and set the value to the localized name of the bundle.

In determining when to display localized names for bundles, Mac OS X prefers user-customized names over the default or localized names stored in the bundle. The `Info.plist` file in the bundle's Contents directory contains the CFBundleDisplayName key, whose value is the display name of the bundle. If the value in this key does not match the name of the bundle in the file system, Mac OS X uses the file system name as the display name. If the values do match, Mac OS X uses the appropriate localized name if it exists; otherwise, it uses the default name.

The rules for supporting localized bundle names applies to all bundles, including applications and frameworks. For more information on bundles, see "Bundles" (page 115).

Localizing Directory Names

If your application package installs any custom support directories, you can provide localized names for those directories as well as for your application. Localization of your application-specific directory names is not required, and may not be practical in all cases. If you do want to localize your application's support directories, you should do so only for directories whose names are known in advance by your application. Localizing user-specified directory names is not recommended.

To localize a directory name, you must add the extension `.localized` to the directory name and mark the extension as hidden by default. Inside your directory, you then create a subdirectory called `.localized`. Inside of this subdirectory, create a strings file for each localization you wish to support. The strings file contains a single entry with the localized version of the directory name. For example, a localized Release Notes directory with English, Japanese, and German localizations would have the following structure:

```
Release Notes.localized/
    .localized/
        en.strings
        de.strings
        ja.strings
```

Inside each of the strings files, you would map the non-localized directory name to the localized name. For example, to map the name "Release Notes" to a localized directory name, each strings file would have an entry similar to the following:

```
"Release Notes" = "Localized name"
```

Note: Many system-defined directories do not include the `.localized` extension on their names. Because these directories existed prior to the introduction of localized file system names, a different localization technique is used to alleviate the need to change these directory names. For these known directories, Mac OS X instead looks for an empty file with the name `.localized` just inside the directory. If this file exists, Mac OS X displays the localized directory name text.

Differences Between HFS+ and UFS

There are many significant differences between the two major file systems on Mac OS X: HFS+ and UFS. In many cases, these differences have some bearing on programs developed for Mac OS X. The following list summarizes the major differences between these file systems (many of these statements apply to HFS as well as HFS+):

- **Case sensitivity**. UFS is sensitive to case; although HFS+ is case-insensitive it is case-preserving.
- **Multiple forks**. HFS+ supports multiple forks (and additional metadata) whereas UFS supports only a single fork. (Carbon simulates multiple forks on file systems that do not support them, such as UFS.)
- **Path separators**. HFS+ uses colons as path separators whereas UFS follows the convention of forward slashes. The system translates between these separators.
- **Modification dates**. HFS+ supports both creation and modification dates as file metadata; UFS supports modification dates but not creation dates. If you copy a file with a command that understands modification dates but not creation dates, the command might reset the modification date as it creates a new file for the copy. Because of this behavior, it is possible to have a file with a creation date later than its modification date.

- **Sparse files and zero filling**. UFS supports sparse files, which are a way for the file system to store the data in files without storing unused space allocated for those files. HFS+ does not support sparse files and, in fact, zero-fills all bytes allocated for a file until end-of-file.
- **Lightweight references to file-system items**. See "Aliases and Symbolic Links" (page 180).

In addition, the APIs historically associated with each file system sometimes have different behaviors. For example, a program using BSD (or BSD-derived) APIs can delete a file that is open; on the other hand, a Carbon program may only delete a file that is closed.

Aliases and Symbolic Links

Aliases and symbolic links are lightweight references to files and folders. Aliases are associated with Mac OS Standard (HFS) and Mac OS Extended (HFS+) volume formats; symbolic links are a feature of UFS file systems. Both aliases and symbolic links allow multiple references to files and folders without requiring multiple copies of these items. Prior to Mac OS X 10.2, aliases and symbolic behaved very differently when a referenced file or folder moved or changed.

Originally, aliases located a file or folder using its unique identity first and its pathname second. If you moved a file on the same volume, any aliases pointing to that file would still point to the original file. If you deleted a file and replaced it with an identically-named file, aliases would still work because they could locate the file by its pathname. Beginning with Mac OS X 10.2, aliases now reverse this search order by using the pathname first and unique identity second.

Because aliases and symbolic links both use a file system path to resolve a file's location, they now offer a similar initial behavior. If you replace a file with an identically-named file, moving the old file to a new location, both aliases and symbolic links point to the new file. However, if you move a file without replacing it, symbolic links to the file break while aliases do not.

On HFS and HFS+ file systems, each file and folder has a unique, persistent identity. Aliases store this unique identity along with the pathname information. If the file cannot be found by its pathname, the alias attempts to locate the file using its unique identity. If it finds the file, the alias updates its internal record with the new path information. Similarly, if the pathname is correct, but the unique identity is wrong, the alias updates its internal record with the unique identity of the new file.

The Finder and other system applications now use aliases with this pathname-first behavior. However, applications can still resolve aliases by unique identity first using the methods of the Alias Manager.

If your application supports versions of Mac OS X prior to Mac OS X 10.2, you should follow certain guidelines when modifying files. First, when editing a file, modify the existing file. Second, if you need to replace a file transparently with a new version, use `FSExchangeObjects` to swap the new file for the old one. The NSDocument class already uses similar techniques to update the document file, thus maintaining aliases whenever possible.

Resource Forks

Before Mac OS X and Carbon, application resources were put in the resource fork of the application executable. That policy has now changed. In Mac OS X and for Carbon applications generally, resources should be put in the data fork of a separate resource file, not the resource fork of the executable.

The Carbon APIs now read and process resources in a resource file’s data fork as if they were in the resource fork. (In fact, the system routines that read resources—which are primarily Resource Manager functions—now do most of the work for you.) If application resources are stored in the resource fork, you can use these APIs to access them, but now you must explicitly specify the resource fork in order for this to happen.

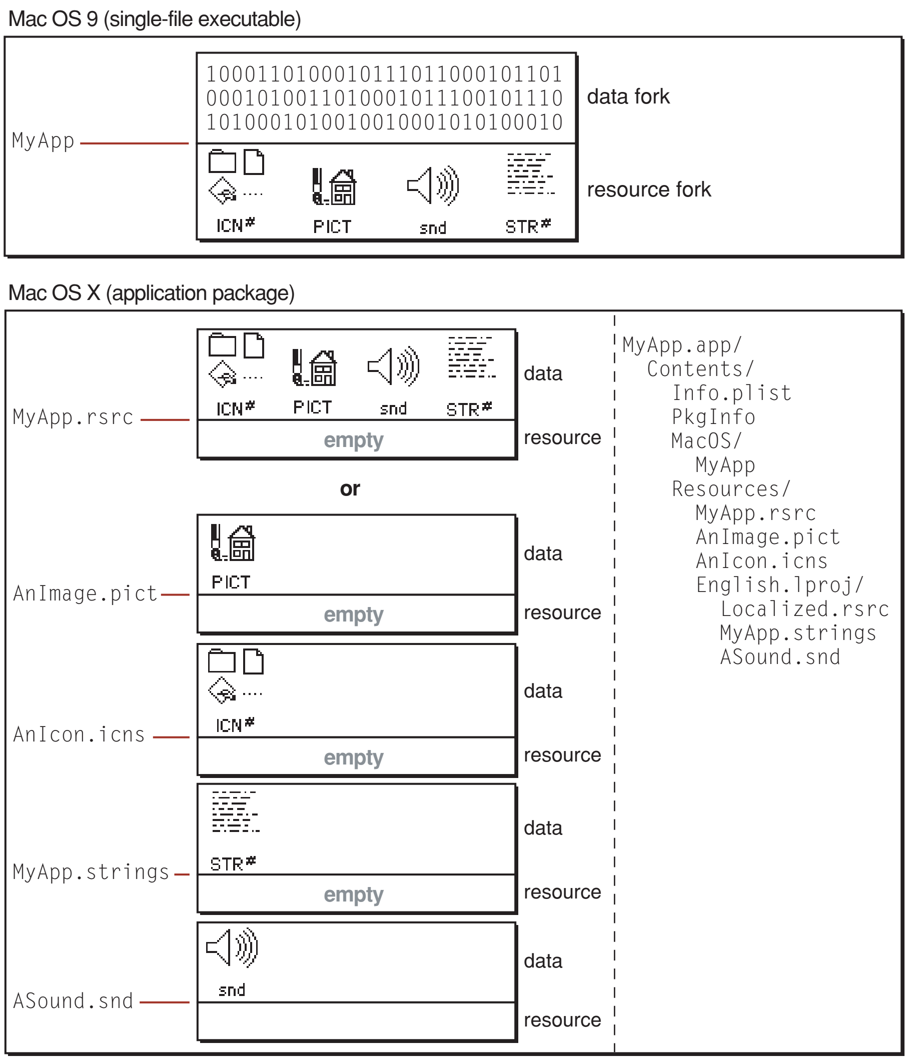
The primary reason for moving application resources out of resource forks is to enable applications to be seamlessly moved around different file systems without loss of their resources; this would include methods such as BSD commands, FTP, email, and Windows and DOS copy commands. Most other computing environments, including the Web, recognize single-fork files only, and tend to delete the resource fork of HFS and HFS+ files.

Even though Apple now recommends storing resources in the data fork of a resource file, this—by itself—is an incomplete solution. For example, application resources stored in a single file are much harder to localize. In addition to moving application resources out of resource forks, you should use the application packaging scheme (see "Application Packaging" (page 131)) and do either of the following:

- In the localized (or nonlocalized) areas of the application bundle, put a file that contains the application resources for that locale (or for all locales). By convention, this file has an extension of `.rsrc`, although it can have any extension or no extension.
- Instead of putting all localized resources in a single `.rsrc` file, put each resource (or groups of related resources) in its own file.

Figure 9-1 depicts how resources can be stored in Mac OS X in contrast to the way they are stored on earlier Mac OS systems.

Figure 9-1 Resources in the data fork



Although Apple supports the all-resources-in-one-file model, it strongly recommends that developers put their resources in separate files. One consideration behind this is the emerging use of XML as a way to specify resources. Carbon has an XML-based runtime that tools such as Interface Builder use to export user interfaces as XML.

As with applications, documents on Mac OS X should have their resources put in the data fork. The reasons for this are the same as the reasons for having application resources in the data fork. It makes it possible to exchange these documents, without loss of resource data, between Macintosh and non-Macintosh systems, including most Web servers.

Files residing on HFS and HFS+ file systems have their Finder attributes stored in a private fork separate from both resource and data forks. These attributes include type and creator codes. Mac OS X maintains these attributes because they enable the Finder to enhance the user's experience. At the same time, however, Apple strongly encourages developers to use file extensions as alternative means for identifying document types. Mac OS X does a very good job of recognizing and handling document extensions. And, as "Copy and Move Operations" (page 195) in the chapter ""The Finder" (page 187)" makes clear, if you copy an HFS or HFS+ document to a different platform (including Web servers), file extensions help ensure that the document's type information is preserved.

File Encodings and Fonts

Although Unicode is considered the native encoding for Mac OS X, there is no file encoding that is the default for all situations. The file encoding that is (or should be) used depends on what you want to do, on the API you use, and on the underlying file system.

For example, the encoding used for filenames differs among the various file systems. Mac OS Extended (HFS+) uses one particular form of Unicode for filenames: canonically decomposed Unicode 2.1 in UTF-16 format (a sequence of 16-bit codes). The UFS file system uses a different form of Unicode for filenames; it allows any character from Unicode 2.1 or later, but uses UTF-8 format (a sequence of 8-bit codes). And Mac OS Standard (HFS) uses legacy Mac encodings, such as

MacRoman. Note that, because of implementation differences, erroneous Unicode in filenames on HFS+ volumes displays correctly on Mac OS 9 systems, but always appears with garbled characters on Mac OS X.

In addition, all code that calls BSD system routines should ensure that the `const *char` parameters of these routines are in UTF-8 encoding. All BSD system functions expect their string parameters to be in UTF-8 encoding and nothing else. An additional caveat is that string parameters for files, paths, and other file-system entities must be in *canonical* UTF-8. In a canonical UTF-8 Unicode string, all decomposable characters are decomposed; for example, é (0x00E9) is represented as e (0x0065) + ´ (0x0301). To put things in canonical UTF-8 encoding, use the "file-system representation" APIs defined in Cocoa and Carbon (including Core Foundation). For example, to get a canonical UTF-8 character string in Cocoa, use NSString's `fileSystemRepresentation:` method; for noncanonical UTF-8 strings, use NSString's `UTF8String` method.

If you use regular QuickDraw and want to draw text, you should be aware of some potential problems. The Carbon File Manager has some file-system calls that return Mac encodings and others that return Unicode. If you get Unicode text, you will have problems drawing it using QuickDraw Text because that API doesn't directly support Unicode. On the other hand, if you get a Mac encoding and you want to use Cocoa or Carbon's Apple Type Services for Unicode Imaging (ATSUI) APIs, you must convert it to Unicode first.

Generally, the encoding that is used depends upon the API you use and not the font. Fonts are not necessarily limited to particular encodings. TrueType fonts, for example, declare the set of glyphs they implement and provide encoding tables that map those glyphs to character values in particular encodings. PostScript fonts have similar encoding tables. Various parts of the operating system know how to map characters from one encoding to another. Cocoa and ATSUI use Unicode as the "destination" mapping for a font. QuickDraw Text in Carbon uses the Mac encodings, selected according to the script that the 'FOND' resource of the font corresponds to.

The fonts that are installed with Mac OS X have large character sets supporting a wide range of encodings and scripts. For example, Lucida, the system font, supports extended Latin, Greek, Cyrillic, Arabic, Hebrew, and Thai. But if you draw text through QuickDraw Text, you have access only to the MacRoman repertoire. To access the rest, you must use Cocoa or ATSUI. Similarly, the Hiragino fonts also have a large repertoire of characters beyond that supported by MacJapanese, and these are accessible only through Cocoa or ATSUI. Both Cocoa and ATSUI also substitute glyphs from other fonts when the requested one isn't available; however, their algorithms for font substitution are different.

For information on file encodings in the context of multiscript support, see "Adding Multiscript Support" (page 220) in the chapter ""Internationalization" (page 209)."

C H A P T E R 1 0

The Finder

The Finder is the primary application of Mac OS X. Running from the moment you log in, it works with the system software to track and manage the Dock, the file system (including mounted network volumes), and connected devices. Through the windows of the Finder, users can view and manipulate items in the file system such as folders, applications, and documents.

This chapter does not go into detail about the human-interface elements related to the Finder. Instead it focuses on those aspects of the Finder that are of special interest to Mac OS X software developers. This information includes

- application interfaces to the Finder
- the stores of information the Finder maintains
- how the Finder handles applications and documents
- how the Finder handles file operations that take place between volumes of different formats
- how the Finder determines which application to launch when the user double-clicks a document
- how the Finder handles file sorting

The Role of the Finder

In general, the nature and role of the Finder in Mac OS X is much the same as it is in Mac OS 9. The Finder in Mac OS X is an application—specifically, a Carbon application—that manages the user's desktop and mediates user access to applications, documents, and other items in the file system. Users launch applications and open documents through the agency of the Finder. In a sense, it is the primary application, the one that is constantly running while users are logged in to the system.

There are, however, several striking differences in Mac OS X that affect the nature and role of the Finder:

- **The Aqua human interface**. This interface affects not only the presentation of desktop elements, but the logic and mechanics behind their use. The Dock and the controls of Finder windows, for example, introduce paradigms absent from Mac OS 9.
- **Multiple users**. Yes, Mac OS 9 supports multiple users (through the Multiple Users control panel), but there it is an option. On a Mac OS X system, multiple users is standard. Users must log in to a Mac OS X system (even if they request logging in to happen automatically). Once logged in, they work in an environment largely customized to their own specifications.
- **Multiple application environments**. Again, the difference in this respect is not absolute; if you take Java into consideration, Mac OS 9 does (or can) have multiple application environments. However, the difference in degree is significant. Mac OS X must deal with the Carbon, Cocoa, Java, Classic, and (in some cases) the BSD Commands application environments.
- **Multiple volume formats**. Mac OS X supports various volume formats, both multiple-fork formats such as Mac OS Extended (HFS+) and flat-file formats (UFS, among others). It tries to make all file-system operations between volumes of different formats as seamless as possible. See "The Finder and File Operations" (page 195) for further information.

The Finder attempts to make the user's experience of all application environments as much the same as possible. However, there are a few issues with the Classic environment. Classic applications cannot run from volumes that are not Mac OS Standard (HFS) or Mac OS Extended (HFS+). Applications from all other environments can run from any volume, regardless of format. In the same vein, Classic applications cannot open or save documents on any volume that is not HFS or HFS+. For more on the Classic environment, see "The Classic Environment and Your Application" (page 239).

Finder Interfaces to Applications

At present, the Finder offers the information property list as an interface for applications. Through this interface, applications can communicate their essential data to the Finder.

The Finder also defines a suite of Apple events that applications can use to accomplish a number of tasks, including opening documents and launching applications.

Apple Events

Applications can send Apple events to the Finder to accomplish a number of operations. These operations include

- launching other applications
- opening URLs
- opening documents

Cocoa provides an object-oriented mechanism for communicating Apple events between the Finder and an application. You can send messages to a shared instance (one per application) of the NSWorkspace class and that object handles the composition and sending of Apple events to the Finder. When the Finder sends Apple events to a Cocoa application, the Application Kit converts them to messages to the application delegate object.

Information Property Lists

When you develop an application or any other bundle for Mac OS X, you must specify as part of the project certain key-value pairs for the bundle's information property list. This property list is in a file named `Info.plist` that, when the application is built, is made part of the bundle. It contains the following Finder-specific information:

- name of application (displayed by the Finder)

- type and creator codes (type is `'APPL'` for applications)
- icon filename
- version string
- descriptive information (displayed by the Finder)
- documents handled by this application, including document name, icon, role, types, and extensions
- URLs handled by this application, including URL name, icon, and schemes

The use of property lists to convey information is more passive than other interfaces; a developer need only make this information available to the Finder. When the Finder encounters an application, it extracts the information in `Info.plist` and uses it to populate its databases (see "Information Stored by the Finder" (page 190) for details).

For more information on information property lists and the keys that are specific to the Finder, see "Information Property Lists" (page 198) in the chapter ""Software Configuration" (page 197)." For related information, see the chapters "Bundles" (page 115) and "Application Packaging" (page 131).

Information Stored by the Finder

The Finder maintains a number of (private) databases that give it a comprehensive, if not entirely complete, view of the desktop, applications, documents, and other items that are part of the user experience. This section describes how the Finder populates these databases and gives some idea of the information that resides in them. It also describes file attributes of specific interest to the Finder.

Collecting Application Information

The way that the Finder in Mac OS X stores file-system information differs from the way the Finder in Mac OS 9 stores that information. The Finder in Mac OS 9 associates a desktop database with each mounted volume on the system. Each database contains information about all files and directories on the volume. When the system is booted, the Finder builds these databases and, thereafter, dynamically updates them as files and directories are added, changed, and removed.

In Mac OS X the situation is different. Because of the multi-user nature of Mac OS X, the Finder maintains an application database for each user who has an account (local or network) on a system. This database contains information about all the applications the Finder has encountered for that user and includes information about the document types understood by each application. The Finder works with Launch Services to extract this information from the information property lists of applications (see "Information Property Lists" (page 189) for a summary of this information).

The way the Finder in Mac OS X builds its databases is also different from the Finder in Mac OS 9.

- The Finder first adds applications at boot time by scanning the standard locations for applications in the user, local (plus system), and network domains.
- When users navigate through the file system, the Finder adds applications in each visited directory to its databases.
- When users try to open a document or attempt any other action that requires an application, and the Finder cannot find an appropriate application, it displays a dialog, allowing the user to select an application. This application is added to the user's application database.

Because there may be locations in the file system a user has never visited, or documents of a type the user has never attempted to open, the Finder might have an incomplete view of the applications available on a system. Yet it has a built-in capability for "lazily" updating its view of the file system.

The Desktop Folder

Both the Mac OS 9 Finder and the Mac OS X Finder store the contents of the user's desktop in a folder. The folder location and desktop semantics for this folder are as follows for the two operating systems.

- In Mac OS 9, an invisible directory called `Desktop Folder` is at the root of each volume; in Mac OS X a visible `Desktop` directory is in each user's home directory (`~/Desktop/`).

- In Mac OS 9 what is displayed on the desktop is the union of `Desktop Folder` directories from all volumes; in Mac OS X, what is displayed on the desktop is the contents of the `Desktop` directory in the logged-in user's home directory.

Finder Attributes

Finder attributes (also known as Finder Info) can be associated with files and folders in the Mac OS X file system. These attributes affect how the Finder displays or handles these files and folders. The Finder in Mac OS X recognizes fewer such attributes than the Finder in Mac OS 9. The supported attributes include

- bundle bit
- invisible bit
- type and creator codes
- custom icons

The attributes not supported in Mac OS X are

- icon position
- view type
- label

In Mac OS X the Finder stores attributes in an invisible per-folder file that contains a data structure that is extensible and volume-format "agnostic."

The Handling of Applications and Documents

The Finder is the user's primary interface to applications and documents in Mac OS X. The following sections describe the techniques used by the Finder to display documents and respond to user interactions with those documents.

Presenting Documents

As described in "Information Stored by the Finder" (page 190), the Finder collects information from applications in the file system and populates a number of databases with that information. When the Finder encounters a file or folder, it often uses this information to determine how to present the file or folder and how to manage the user's interaction with it.

The Finder uses a combination of bundle bit, type code, creator code, and filename extension information to identify and appropriately handle documents, applications, and loadable bundles. The following steps outline the general logic of the Finder when it comes across an item in the file system:

1. Determine whether it is a file or a folder.

   If it is a folder, the Finder determines if it is a bundle (step 2); if it is a file, it determines the kind of file (step 4).

2. Determine whether the folder is a bundle or a regular folder.

   The Finder uses either the bundle bit or the folder extension to determine if a folder is a bundle. The presence of the bundle bit is not necessary and, in fact, the system frameworks provided by Apple do not have the bundle bit set.

3. Find out the type of bundle.

   The Finder obtains the type and creator codes from the information stored within the bundle (see "Anatomy of a Bundle" (page 117) in the chapter ""Bundles" (page 115)"). From the type code (or from the extension if the type code is not available) it determines the kind of bundle. Unless the bundle is a framework, it treats the bundle as a file (in other words, it is a file package).

4. Determine whether the file (including file packages from step 3) is an application.

   If the file is a bundle, and the bundle is an application (as determined by type code or extension), the Finder hides the `.app` extension, if it exists. The Finder adds the information in the application's information property list to its application database for the user (if such information is absent from the database); this is described in "Collecting Application Information" (page 190). If the file is not an application, it is a document (step 5).

5. Display the document appropriately.

   The Finder consults the application database and locates the icon to display next to the filename. If no such icon exists, it displays the default document icon.

Two additional features that affect the way the Finder presents files to the user are filename extension hiding and filename localization. These features are cosmetic additions to the Aqua interface that alter the displayed name of a file without changing the actual name of the file in the file system. The Finder uses routines provided by Launch Services to obtain a displayable name for each file. For information on filename extension hiding, see "Hiding Filename Extensions" (page 175). For information on localized filenames, see "Localizing File System Names" (page 176)

Sorting Files

Mac OS X provides many ways for users to sort and organize documents using the Finder. As in Mac OS 9, users can sort files by name, by size, by modification date, and so on. However, Mac OS X does differ from Mac OS 9 in the way it sorts text strings. In Mac OS X, sorting is based on the Unicode Collation Algorithm (Technical Standard UTS #10) defined by the Unicode Consortium. This standard provides a complete and unambiguous sort ordering for all Unicode characters and is available on the Unicode Consortium website (http://www.unicode.org).

The Finder in Mac OS X takes advantage of some sanctioned ways for altering the default sorting behavior defined by the Unicode standard. In particular, the Finder supports the following sorting rules:

- Punctuation and symbols are significant for sorting.
- Digit sub-strings are sorted by numeric value rather than as characters.
- Case is insignificant.

Choosing Applications to Launch

When a user double-clicks or otherwise tries to open a document in the file system, the Finder checks the document's type and creator codes (if it is an HFS or HFS+ file) or the document's file extension. It uses this information to obtain a list of applications that claim the document type and then open the document with the appropriate application. The following list shows the rules followed by the Finder to determine the appropriate application:

1. Select the application chosen by the user to open the document.
2. Select the application chosen by the user to handle documents similar to this document, meaning documents with the same creator, type, and extension.

3. Select the default application that is registered to handle documents with the same type or extension.
4. Select the application that claims documents of this type.

   The Finder gives preference to native applications over Classic applications. It also gives preference to later versions of an application and to applications with a later modification date.
5. If there is an application with the same creator type already running, select that application instead of the current selection.
6. If the Finder still cannot find a suitable application, it puts up a dialog asking the user to select an appropriate application.

   If the user chooses an application, the Finder adds that information to its application database for future reference.

Once a suitable application has been found, the Finder proceeds to launch the application and pass it a reference to the document.

The Finder and File Operations

The Finder is the "traffic manager" for most if not all file operations that take place in Mac OS X. Unless you use shell commands such as `cp` and `mv` (something generally not recommended), or AppleScript, or some other programmatic means, you must use the Finder to copy, move, and delete files, as well as to make aliases. Obviously, there are issues related to these operations that relate to multiple volume formats. This section discusses how the Finder manages file operations across volumes of different formats.

Copy and Move Operations

When the Finder copies or moves a file, it uses the richest model available, given the formats of the source and destination volumes. The formats that are most significant in these kinds of operations are HFS+ (or HFS) and UFS. These operations particularly affect the representation of the HFS and HFS+ resource fork and the Finder attributes, especially the type and creator codes.

As one might expect, the Finder preserves the resource fork and Finder attributes of an HFS+ file "as is" when it copies the file to an HFS+ (or HFS) volume. The more interesting case, however, is when it copies an HFS+ file to a UFS volume. When this happens, the Finder splits out the information that is *not* in the data fork (particularly the type and creator codes) and writes this information to a hidden file in the same directory location as the copied file. This hidden file has the same name as the UFS file, except that it has a "dot-underscore" prefix. Thus, if you have an HFS+ file named `MyMug.jpeg`, when you copy it to a UFS volume, there will be a file named `._MyMug.jpeg` in the same location.

When the Finder copies a UFS file to an HFS or HFS+ volume, it looks for the hidden "dot-underscore" file. If one exists, it creates an HFS+ (or HFS) file, using the information in the hidden file to recreate the file's resource fork and Finder attributes. If the hidden file does not exist, the copied file has no resource fork.

Note that the Finder accomplishes these operations through the Carbon APIs on which it is based.

Note: You can use the BSD `cp` or `mv` commands on a application package (or any other bundle) without ill effect. However, if you use those commands on a single-file CFM application, the copied (or moved) application is rendered useless. For the latter purpose, Apple includes the `CpMac` command-line utility.

Management of Aliases and Symbolic Links

Mac OS Standard (HFS) and Mac OS Extended (HFS+) file systems include the file system entity known as an alias. An alias bears some similarities to a symbolic link in a UFS file system, but the differences are significant. See the section "Aliases and Symbolic Links" (page 180) in the chapter ""The File System" (page 163)"for a description of these differences.

How the Finder manages a file-system world in which both aliases and symbolic links coexist is simple. It recognizes symbolic links but creates only aliases (when given the appropriate menu command). Even when it encounters a symbolic link in the file system, it presents it as an alias—that is, there is no visual differentiation between the two. The only way to make a symbolic link in Mac OS X is to give the BSD command `ln -s`.

C H A P T E R 1 1

Software Configuration

Mac OS X gives you a number of ways to configure your software. It stores all configuration data persistently using various mechanisms. These mechanisms permit dynamic updating of this data and make it available to programs at runtime.

Mac OS X has three basic configuration options:

- **Property lists**. A textual way to represent data, using XML as the structuring medium. Elements of the property list represent data of certain types, such as arrays, dictionaries, and strings. System routines allow programs to read property lists into memory and convert the represented data into "real" data.
- **Information property lists**. A special form of property list with predefined keys for specifying basic bundle attributes and information of interest to the Finder and other applications. The information property list is stored inside a bundle. It specifies information such as supported document types, URL schemes, and copyright and version information. The information property list also allows the specification of user-defined keys.
- **Preferences system**. Allows you to create, write, and read preferences per user, per application, and per host.

Property Lists

A property list in Mac OS X is a textual representation of data that uses the Extensible Markup Language (XML) as the formal structuring medium. The flexibility that such structuring affords is a great programmatic convenience.

(See http://www.w3.org/XML/1999/XML-in-10-points for an excellent summary of XML.) Elements of XML correspond to programmatic entities such as arrays, dictionaries, and strings.

You can create a property list with the Property List Editor application or, if that is not available, any text editor. Then you add the file to your project. Property lists are stored as a bundle resource (usually nonlocalized). Once your program is built and run, it can easily access the information in the property list by using special routines that read the property list and convert the data represented in it to the appropriate types. The supported property-list types are dictionary, array (vector), string, data, date, number, and Boolean.

Custom property lists are sometimes used to specify certain types of initialization data, such as key bindings. A file named `CustomInfo.plist` is often used for this purpose.

Information Property Lists

Information property lists are system property lists (see "Property Lists" (page 197)) that contain essential configuration information for bundles. This information is readily available to system and program code at runtime. As described in the section "Types of Bundles" (page 122) of the chapter ""Bundles" (page 115)," a bundle is a packaging scheme and generic programmatic type for such things as applications, frameworks, and plug-ins. Information property lists are thus a pervasive and important means for configuring software of almost all kinds. They make available information that the Finder (and possibly other applications) need, and they enable applications to deal with HFS and HFS+ files.

By convention, information property lists are found in files with the name `Info.plist`. They can contain platform-specific information, in which case the tag for the platform is embedded in the filename; the standard platform-specific names are the following:

```
Info-macos.plist
Info-macosclassic.plist
```

If the configuration information is generic to all platforms (as is ideally the case), the name is `Info.plist`. When the bundle code is executed, it looks first for the platform-specific file; if that does not exist in the bundle, it reads the platform-generic file. Because the search algorithm searches for a file and not a particular key, if you have both a platform-specific file and a platform-generic file, make sure each contains a corresponding set of key-value pairs. Information property list files are located in the `Contents` directories of bundles.

The `Info.plist` file for a bundle can contain all kinds of information. At the top level of the property list, this information is specified as key-value pairs (that is, as a dictionary). Mac OS X defines a set of standard keys for basic configuration information, such as the name of the executable and the version of the bundle. The Finder also defines keys for such things as documents, icons, and the information it displays to users. You are free to define and use your own keys. The integrated development environment (IDE) provides the human interface for entering standard, Finder, and custom configuration data in the `Info.plist` file as key-value pairs. For the standard and Finder-specific information property list keys, see Appendix A (page 273).

A special localized resource file named `InfoPlist.strings` goes with the `Info.plist` file. The `InfoPlist.strings` file contains keys for the information property list that might need to be localized. These keys are the values specified for associated keys in the `Info.plist` file. Commonly localized keys are CFBundleName, CFBundleShortVersionString, CFBundleGetInfoString, CFBundleGetInfoHTML, and the values of the CFBundleTypeName and CFBundleURLName types. See "Bundles" (page 115) for more about localized bundles, particularly where they go in the bundle and how they are located.

Document Configuration

Information property lists for applications that create or "understand" documents permit the definitions of abstract types and roles. These definitions apply to Clipboard (pasteboard) data as well as documents.

An abstract type defines general characteristics of a family of documents. Each abstract type has corresponding concrete types, such as a filename extension or a 4-byte identifier. Concrete types are ways that an abstract type is encoded in various file systems or persistent formats. The notion of abstract types improves general application interoperability by removing the current dichotomy between the pasteboard type system and the filename-extension type system. Abstract type names should have a copyright to ensure uniqueness.

A role defines an application's relation to a document type. There are five roles:

- **Editor**. The application can read, manipulate, and save the type.
- **Viewer**. The application can read and present the data.
- **Printer**. The application can print the data only.
- **Shell**. The application provides runtime services for other processes—for example, a Java applet viewer. The name of the document is the name of the hosted process (instead of the name of the application), and a new process is created for each document opened.
- **None**. The application does not understand the data, but is just declaring information about the type (for example, the Finder declaring an icon for fonts).

An Example of an Information Property List

Listing 11-1 contains an example of an `Info.plist` file. This information property list, which is based on the Sketch demonstration application, demonstrates additional keys not found in the default Sketch application.

Listing 11-1 The `Info.plist` file for the Sketch demo application

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
    <key>CFAppleHelpAnchor</key>
    <string>sktch001</string>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleName</key>
    <string>Sketch</string>
    <key>CFBundleDisplayName</key>
    <string>Sketch</string>
    <key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>sketch</string>
            <string>draw2</string>
        </array>
        <key>CFBundleTypeIconFile</key>
        <string>Draw2File</string>
        <key>CFBundleTypeName</key>
        <string>Apple Sketch Graphic Format</string>
        <key>CFBundleTypeOSTypes</key>
        <array>
            <string>sktc</string>
        </array>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>NSDocumentClass</key>
        <string>SKTDrawDocument</string>
        <key>NSExportableAs</key>
        <array>
            <string>NSPDFPboardType</string>
            <string>NSTIFFPboardType</string>
        </array>
    </dict>
    <dict>
        <key>CFBundleTypeExtensions</key>
        <array>
            <string>pdf</string>
        </array>
        <key>CFBundleTypeName</key>
        <string>NSPDFPboardType</string>
        <key>CFBundleTypeOSTypes</key>
        <array>
            <string>pdf </string>
        </array>
        <key>CFBundleTypeRole</key>
        <string>None</string>
    </dict>
    <dict>
        <key>CFBundleTypeExtensions</key>
        <array>
            <string>tiff</string>
        </array>
        <key>CFBundleTypeName</key>
        <string>NSTIFFPboardType</string>
```

```
            <key>CFBundleTypeOSTypes</key>
            <array>
                <string>tiff</string>
            </array>
            <key>CFBundleTypeRole</key>
            <string>None</string>
        </dict>
    </array>
    <key>CFBundleExecutable</key>
    <string>Sketch</string>
    <key>CFBundleIconFile</key>
    <string>Draw2App</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.CocoaExamples.Sketch</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleSignature</key>
    <string>sktc</string>
    <key>CFBundleVersion</key>
    <string>1.2.0</string>
    <key>NSAppleHelpFile</key>
    <string>osxa444.htm</string>
    <key>NSAppleScriptEnabled</key>
    <string>YES</string>
    <key>NSJavaNeeded</key>
    <string>YES</string>
    <key>NSJavaPath</key>
    <array>
        <string>Sketch.jar</string>
    </array>
    <key>NSJavaRoot</key>
    <string>Contents/Resources/Java</string>
    <key>NSMainNibFile</key>
    <string>Draw2Java.nib</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
</dict>
</plist>
```

The Sketch application also includes a localized version of several keys in an `InfoPlist.strings` file. Listing 11-2 shows the English version of this file.

Listing 11-2 The `InfoPlist.strings` file for the Sketch demo application

```
// InfoPlist.strings
// Sketch Exampmle

{
    CFBundleName = "Sketch";
    CFBundleGetInfoString = "Apple Sketch Application Example 1.2.0. 
Copyright \U00A9 1998-2001, Apple Computer, Inc.";
    NSHumanReadableCopyright = "Copyright \U00A9 1998-2001, Apple Computer, 
Inc.";
    // Document type human-readable names.
    "Apple Sketch Graphic Format" = "Apple Sketch Graphic Format";
    "NSPDFPboardType" = "Portable Document Format (PDF)";
    "NSTIFFPboardType" = "Tagged Image File Format (TIFF)";
}
```

The Preferences System

Preferences are application or system options that allow users to customize their working environment. Most applications read in some form of user preferences. For example, a document-based application may store preferences for the default font, automatic save options, or page setup information. Preferences are also not limited to applications. You can read and write preference information, including user preferences, from any frameworks or libraries you define.

The preferences system of Mac OS X includes built-in support for preserving and restoring user settings across sessions. Both Carbon and Cocoa applications can use Core Foundation's Preference Services for reading and writing preference information. Cocoa applications can also use the NSUserDefaults class to read user preferences.

Important
The assumption with user preferences is that they are not critical; if they are lost, the application should be able to recreate the default set of preferences. You should not store an application's initial configuration data as a preference. Initial configuration data *is* critical and should be stored in the information property list, or some other property list stored, inside the application package.

The preferences system stores values that are associated with a key; later you can use the key to look up the preference value when you need it. Key-value pairs are assigned a scope using a combination of user name, application ID, and host (computer) name. This mechanism allows you to create preferences that apply to different classes of users. For example, you can save a preference value that applies to

- the current user of your application on the current host
- all users of your application on a specific host connected to the local network
- the current user of your application on any host connected to the local network (the usual category for user preferences)
- any user of any application on any host connected to the local network

Applications should store only those preferences that represent information captured from the user. Storing the same set of default preferences for each user is an inefficient way to manage your application's preferences. Preferences are stored in property list files that must be parsed to read in the preference information. A more efficient way to manage preferences is to store a set of default preferences internally and then apply any user-customized preferences on top of the default set.

How Preferences Are Stored

The preferences system stores preference data in files located in the `Library/Preferences` folder in the appropriate file-system domain. For example, if the preference applies to a single user, the file is written to the `Library/Preferences` folder in the user's home directory. If the preference applies to all users on a network, it goes in `/Network/Library/Preferences`.

Each of the files in `Library/Preferences` takes a name that uniquely identifies an application. Each name is from application's bundle identifier. You assign the bundle identifier (using the key CFBundleIdentifier) in your application project as part of its information property list (see "Standard Bundle Keys" (page 274) for details). The system routines related to preferences use the bundle identifier to find the preferences for a given application.

To ensure that there are no naming conflicts, Apple strongly recommends that bundle identifiers be the same form as Java package names—your company's unique domain name followed by the application or library name. Some examples are `com.apple.finder`, `com.apple.Sherlock`, and `com.foo.ImageImport`. Using this scheme minimizes the possibility of name collision and leaves you responsible for managing the identifier name space under your corporate domain.

Core Foundation's Bundle Services and, for Cocoa applications, the NSBundle class provide routines for accessing an application's bundle identifier. You should always use these routines and never hard-code the application identifier.

The preferences files in `Library/Preferences` have the extension of `.plist`. This extension indicates that they contain property lists. If you must edit preferences files, use the Property List Editor application. If you wish, you can modify the XML property lists directly to add or change application preferences. By doing so, however, you can introduce editing errors into the XML data; if this happens, the application might not be able to load the file and thus would lose all its preferences.

Problems might ensue if an application tries to write preferences to a location other than `Library/Preferences` in the appropriate file-system domain. For one thing, the preferences APIs aren't designed for this difference. But more importantly, preferences stored in unexpected locations are excluded from the preferences search list and so might not be noticed by other applications, frameworks, or system services.

Preference Domains

When you create a new preference or search for an existing one, the preferences system uses the notion of preference domains to specify the scope and location of the preference. A preference domain consists of three pieces of information: an application identifier, a host name, and a user name. Table 11-1 shows all of the preference domains, listed in the order they are searched when the preference system attempts to locate a preference value.

Table 11-1 Preference domains in search order

1	Current User	Current Application	Current Host
2	Current User	Current Application	Any Host
3	Current User	Any Application	Current Host
4	Current User	Any Application	Any Host
5	Any User	Current Application	Current Host
6	Any User	Current Application	Any Host
7	Any User	Any Application	Current Host
8	Any User	Any Application	Any Host

The search routines look through the various preference domains in the order given above until they find the key you have specified. If a preference has been set in a less-specific domain—"Any Application," for example—its value is retrieved with this call if a more specific version cannot be found. This means that values in more-specific domains override those for the same key in less-specific domains.

The defaults Utility

The preferences system of Mac OS X includes a command-line utility named `defaults` for reading, writing, and removing preferences (or, user defaults) from application and other domains. The `defaults` utility is invaluable as an aid for debugging applications. Much of the preferences information is accessible through an application's Preference dialog (or the equivalent), but some of it isn't, such as the position of a window. You can access this information with the `defaults` utility.

To run the utility, launch the Terminal application and, in a BSD shell, enter `defaults` plus all appropriate command options. For a terse description of syntax and arguments, run the `defaults` command by itself. For a more complete description, read the man page for `defaults` or run the command with the `usage` argument:

```
$ defaults usage
```

Because applications access the preferences system while they are running, you should not modify the defaults of a running application using `defaults`. If you change a default in a domain that belongs to a running application, the application probably won't see the change and might overwrite the default.

CHAPTER 12

Internationalization

Mac OS X is an internationalized operating system. As such, it not only facilitates the localization of software resources (text, images, sounds, and so on) but it also can display text containing more than one script. However, Mac OS X cannot fully meet the claim of being internationalized without the cooperation of developers. You must play a part and ensure that any software intended for markets outside your own country or region is properly internationalized and localized. You should also ensure that your application supports the presentation of multiscript text.

Before going further, it might be helpful to distinguish the seemingly similar terms localization, internationalization, and multiscript support.

- **Localization** is the adaptation of a software product, including 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 translating user-interface text, resizing text-related graphical elements, and modifying images and sound to conform to local conventions.
- **Internationalization** is the design or modification of a software product to facilitate localization. With its Unicode-based text storage, bundled resources, and preferences system, Mac OS X is an internationalized operating system that enables the input, display, formatting, and manipulation of localized resources. To internationalize software, you must write code that makes use of these locale-aware services.
- **Multiscript support** refers to a set of programming practices that ensures software can appropriately handle multilingual text. A program with such support can, for example, accurately display a single document that contains multiple scripts such as English, Japanese, and Arabic.

Except for user-interface tweaking, this chapter does not cover localization because this discipline falls outside software development. Translating text is typically done by professional translators, and recreating images and sounds is usually done by artists and technicians. Of course, developers often perform these functions themselves, but this work does not require them to modify source code. However, it is important for developers to become involved in the localization effort. For example, they should always verify how the translated text of a user interface fits within each user-interface element and either adjust those elements appropriately or request another translation.

Internationalizing Your Application

Internationalization is the process of making your application language-independent in such a way that a user can choose a version of the application that is localized to any of a number of languages or regions. It is only the resources of the application—its text, images, sounds, and so on—that are localized. The executable used for all localized versions of the application is the same.

Internationalizing your application makes your application localizable. That is, you build localization support into your application by placing the text, images, and sounds specific to a language in files in a language-specific subdirectory of your project directory and by using the proper locale-savvy APIs for accessing those resources.

Even if you don’t have immediate plans to support multiple languages in your application, there are advantages to designing your application with internationalization in mind. If your application is properly designed, you won’t have to touch its source code to introduce future localizations; therefore, you won’t run the risk of introducing bugs by putting the necessary hooks in later. Second, you can test the localization code along with the initial monolingual product, thereby minimizing the amount of testing needed for any future localized version.

The internationalization system of Mac OS X relies on a number of technologies:

- **Unicode**. Mac OS X uses Unicode as its native character encoding because Unicode makes it possible to represent most of the languages of the world. With Unicode, the file systems of Mac OS X have no need to change localizations for any encoding. By being Unicode-based, the operating system makes it possible for a speaker of one language to name a file or volume using a script from another language. However, current text systems do not support some writing systems (vertical and right-to-left), so it's not practical to localize to those languages. Additionally, Mac OS X expects either the UTF-8 or the UTF-16 encoding for Unicode, depending on the level of software. You use the former for routines at the lower levels of the system (BSD, NetInfo, and so on) and you use UTF-16 encoding at higher levels of the system.

A properly internationalized application, because it is Unicode-aware, would also include multiscript support (see "Adding Multiscript Support" (page 220)).

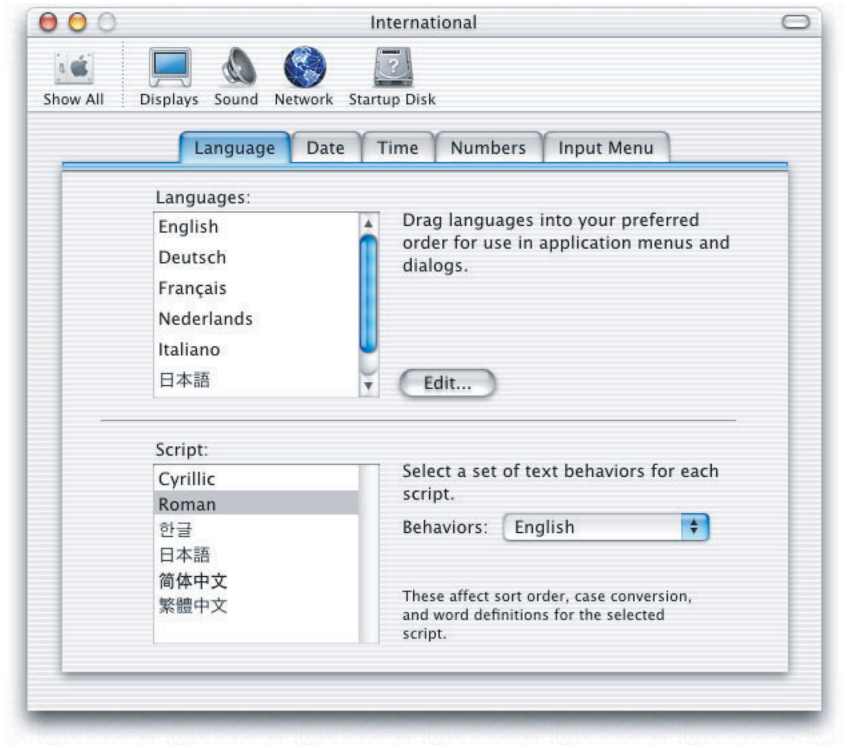
- **Bundles**. Bundles and the APIs related to bundles provide a convenient way to package and access resources localized for a particular language and region. Read the chapter "Bundles" (page 115) for further information.
- **Preferences system**. The preferences system for Mac OS X provides a way to capture and store user preferences, including language preferences. See "Language Preferences and Bundle Resources" (page 211) for details on this set of preferences.

And, of course, the necessary fonts and input services for a particular script must be installed on a Mac OS X system.

Language Preferences and Bundle Resources

The Language pane of the International system preferences lets users set the order of the languages and locales (that is, regional variants of a language) they prefer for their computing environment. The preferences system (described in "The Preferences System" (page 203)) stores this ordered list of languages and locales as a per-user default under the key `AppleLanguages`. Thus a user who understands more than one language can specify alternatives if an application does not include a localization of his primary language.

Figure 12-1 Language pane of the System Preferences International module



Languages:
English
Deutsch
Français
Nederlands
Italiano
日本語

Drag languages into your preferred order for use in application menus and dialogs.

Script:
Cyrillic
Roman
한글
日本語
简体中文
繁體中文

Select a set of text behaviors for each script.

Behaviors: English

These affect sort order, case conversion, and word definitions for the selected script.

Each string in the list of languages potentially corresponds to the base name of an `.lproj` directory in the `Resources` directory of a bundle. The system's bundle-management routines use this string to find the localized resources in the corresponding `.lproj` directory. If that directory does not exist, they use the second language preference to search for a bundle localization. It continues until they find a localization, the default localization being the language used in development.

Core Foundation Bundle Services (CFBundle) provides this search functionality for Carbon and Java applications; for Cocoa applications, it is provided by the NSBundle class. Once your application contains translated versions of language-specific resources, it can load these localized resources from the appropriate set of files, based on the user's language preferences. Thus, your application automatically presents itself to each user in one of that user's preferred languages—ideally (but not necessarily) the user's first choice.

See the chapter "Bundles" (page 115) for information on how bundles store resources.

Designating Languages and Locales

Mac OS X gives you three different ways of expressing language preferences and bundle localizations, and each way carries with it a different degree of specificity. The language designation can be

- a language abbreviation conforming to ISO 639 (for example, "en")
- a locale abbreviation, identifying a regional variant of a language, conforming to ISO 3166 (for example, "en_US"
- a language name (for example, "English")

The recommended approach is to use the ISO 639 language abbreviation or, if appropriate, the ISO 3166 locale abbreviation. However, CFBundle and NSBundle can recognize language names that have a Mac OS language code and can map ISO 639 abbreviations to these names. This lets you specify names such as English, French, German, Japanese, Chinese, Spanish, Italian, Swedish, and Portuguese among others. However, use of these language names is discouraged.

Note: To obtain copies of the ISO 639 and ISO 3166 standards, go to the International Standards Organization website http://www.iso.ch.

Because of the way the system attempts to match language preference with localization, the user's language preferences should be as specific as possible and the localizations stored in a bundle should be as general as possible. As noted earlier, the system bundle routines try to match the designation given for a language preference to the base name of an `.lproj` directory in a bundle. Failing an exact match, it tries to find a localization at a more general level.

An example will help clarify this point. Suppose a user has specified a primary language preference of U.S. English ("en_US"). The user launches an application that has only two localizations, `French.lproj` and `English.lproj`. When the system is asked to fetch a localized resource, it looks first for a directory named `en_US.lproj`, next for directory `en.lproj`, and finally finds the resource in `English.lproj`.

When the bundle routines look for resources, they check both the localization for a single locale and, if the resource isn't found there, the localization for the language of which the locale is a variant. This behavior enables you to put resources that are specific to locales and those that are general to a language in their own `.lproj` directories, and yet have these resources logically combined. The system looks first in a locale-specific localization (say, `en_US.lproj`) and if it doesn't find the resource it's looking for, it looks in a language-specific localization such as `en.lproj` or `English.lproj`. If it can't find it there, it goes on to the user's next preferred language or locale. Apple recommends that applications with a localization for a particular locale should include a localization for the corresponding language; this localization should contain all resources that are generic to the language.

Because they make reference to the abbreviations in the ISO 639 standard, CFBundle and NSBundle can handle most known languages. (To give an idea of how comprehensive their coverage is, the languages include Manx, Faroese, and Oromo.) You can use the Carbon functions in `MacLocales.h` to convert ISO 639 abbreviations to user-visible names in a particular language. You can also use a language or locale abbreviation that is not known to CFBundle and NSBundle; however, if you do, you must ensure that there is an exact match of the abbreviation for the language preference and the `.lproj` directory base name.

Tools for Internationalization and Localization

Internationalizing applications for Mac OS X involves, in part, putting the resources localized for a particular language or region in the proper bundle location. In some situations, you might have to do this task by hand, such as with Code Fragment Manager (CFM) applications (see "CFM Executables" (page 229) for details). Fortunately, for most occasions there are tools to help you.

Project Builder provides the File Reference Inspector to assist you with internationalization. To internationalize a resource file, complete the following steps:

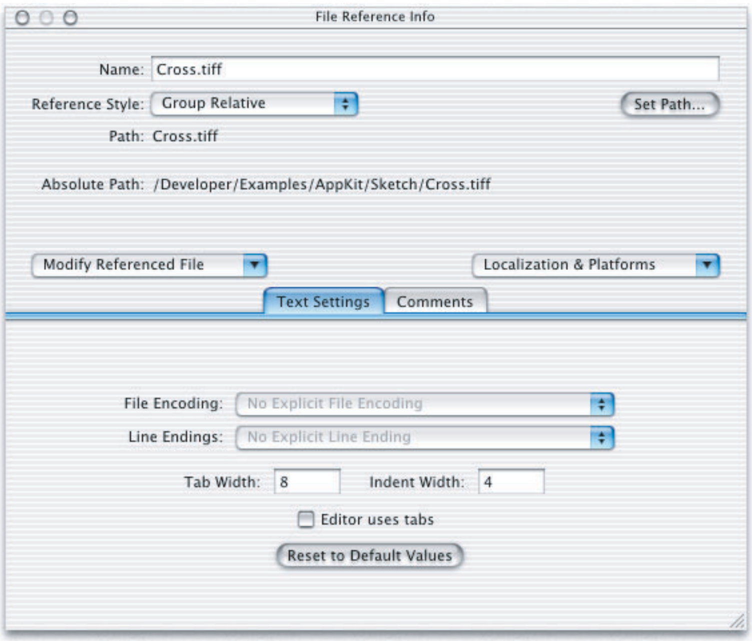
1. Add the resource file to Project Builder.
    a. Select the Resources folder under the Groups and Files pane for a target (you might have to click the Files tab to get the Groups and Files pane).
    b. Choose Add Files from the Project menu.
    c. Use the file browser to navigate to the resource, select it, and click Open.

d. In the dialog that Project Builder displays, select "Copy into group folder", choose one of the "relative" reference styles (for example, Project Relative), and make sure the correct target is selected. Click Add.

2. Select the resource file and choose Show Info from the Project menu.
3. In the File Reference Info window (Figure 12-2 (page 215)), choose Make Localized from the Localization & Platforms pop-up menu.
4. In the dialog that appears, either select one of the standard languages from the pop-up menu or enter a language that is not in the list.

   If you have an existing resource that you want to use as a placeholder or template for another localization, choose Add Localized Variant and select (or enter) the other language. Project Builder copies the resource to the other `.lproj` directory.
5. If the resource is a text file, choose Unicode from the File Encoding pop-up menu.

Figure 12-2 Project Builder's File Reference Inspector


File Reference Info

Name: Cross.tiff
Reference Style: Group Relative
Path: Cross.tiff
Absolute Path: /Developer/Examples/AppKit/Sketch/Cross.tiff

File Encoding: No Explicit File Encoding
Line Endings: No Explicit Line Ending
Tab Width: 8
Indent Width: 4
Editor uses tabs: unchecked

To create "strings" files—which contain localized versions of the strings embedded in your source code—you can use the `genstrings` command-line utility or you can create these files by hand. See "Localizing Strings" (page 217) for more information.

Finally, a word must be said about the tools to use for *localizing* resources. For images and sounds, use the appropriate application (for example, Photoshop). For text, always use a word processor or text editor that can save files in the UTF-16 encoding; the TextEdit application included with Mac OS X provides such capabilities. If you can define and archive user interfaces using an application such as Interface Builder, you should provide localized versions of these interface archives; see the following section for more information on this topic.

Localizing User Interfaces

The native integrated development environment (IDE) for Mac OS X consists of Project Builder, Interface Builder, and a suite of build, debugging, and performance tools. Developers use Interface Builder to create user interfaces for their applications. Interface Builder saves these interfaces as XML archives called nib files. You can localize nib files just as you can localize image and sound files.

Note: This section talks about Interface Builder and its nib files. However, much of the discussion is applicable to localizable user-interface archives created by other IDEs.

Nib files store the user interface of an application, including windows, dialogs, and user-interface elements such as buttons, sliders, text objects, and help tags for these elements. A nib file also holds the connections between these objects that cause actions to be performed when the user activates controls. Nib files are typically localized all at once; the localizer takes a nib file, translates all the user-visible strings and makes other adjustments as necessary (such as resizing the visible elements).

In any medium-size or large application, it's usually a good idea to put each window or panel (that is, dialog) in its own nib file. This practice not only makes it possible to load the user interface lazily (that is, to load it as necessary), but it also permits localization to progress in more incremental steps. It's also a good idea to put the menus of the application in a separate nib file.

You should use Interface Builder to localize nib files. To do this, open all of the nib files in a *language*.lproj directory, localize all the strings, change the sizes of the user-interface elements to accommodate the new strings, and save the nib files. There are a few other things to watch out for:

- Objects in a nib file typically have connections between them that should not be broken. You should lock all connections (an option in Interface Builder preferences) before editing the nibs.
- Numeric and date fields often have formatters attached to them. Change the format, if necessary, to be appropriate for the new locale; an example is the thousands-separator character, which is a comma in the United States and a period in continental Europe. (Consult the Interface Builder documentation for instructions on doing this.)
- Dialogs and windows usually have minimum or maximum sizes that are specified through the inspector. If you must make a dialog or window wider for a given language, it's likely that the minimum size also needs to be modified.
- Some user-interface objects support help tags—bits of explanatory text that appear when the user moves the mouse over the object for a short period. You can define the help tags for an object in Interface Builder's inspector, where they can also be localized.

Localizing Strings

Strings files enable you to externalize and localize the strings that are embedded in your application's source code. They are called strings files because they have the extension of .strings, for example Localizable.strings. There is typically at least one strings file per localization (that is, per .lproj directory) in a bundle.

Note that strings files are not intended for strings that appear in an archived user interface (for example, a nib file created by Interface Builder). For such strings, you can localize them using the appropriate development application (see "Localizing User Interfaces" (page 216)).

Also keep in mind that there are two kinds of embedded strings: those that the user sees, and those that the user doesn't see. An example of a string the user doesn't see is contained in the following statement:

```
if (CFStringHasPrefix(value, CFSTR("-")) {
    CFArrayAppendValue(myArray, value);
};
```

The string “-” does not need to be localized since the user never sees it, and it has no effect on anything that the user does see. On the other hand, a string that appears in an alert dialog should be localized.

Composing a Strings File

Place strings that need to be localized in a strings file, whose format is illustrated below:

```
/* A comment */
"Yes" = "Oui";
"The same text in English" = "Le même texte en français";
```

The string on the left is used as a key in code for locating the string on the right in the strings file. Carbon and Cocoa provide APIs for accessing localized strings from a strings file. Cocoa applications should use the following macros (declared in the header for the Foundation framework’s NSBundle class) to extract strings out of a strings file:

```
NSLocalizedString(key, comment)
NSLocalizedStringFromTable(key, table, comment)
```

Carbon and other non-Cocoa programs should use the equivalent macros defined in Core Foundation Bundle Services (CFBundle):

```
CFCopyLocalizedString(key, comment)
CFCopyLocalizedStringFromTable(key, table, comment)
CFCopyLocalizedStringFromTableInBundle(key, table, bundle, comment)
```

For instance, assuming the French localization was selected

```
NSLocalizedString (@"Yes", @"")
```

would return “Oui” from the above table. The arguments to the above macros are as follows:

key

The string used in looking up the localized value.

table

The name of the strings file to look in (by default, “Localizable”, which causes the macro to look for `Localizable.strings`).

comment

The comment to put in the strings file when generating the strings file.

Some functions and methods (such as the Cocoa `stringWithFormat:` method and the Core Foundation `CFStringCreateWithFormat` function) allow string arguments with formatting characters in the string. For these functions and methods, you can specify formatting characters in both keys and values, as in this example:

```
"Windows must have at least %d columns and %d rows." =
"Les fenêtres doivent être composes de %d colonnes et %d lignes au minimum.";
"File %@ not found." = "Le fichier %@ n'existe pas.";
```

Note: The "%@" specifier is an extension to the standard `printf()` formatting characters. It represents arbitrary Cocoa and Core Foundation objects. See *Inside Cocoa: Object-Oriented Programming and the Objective-C Language* for the complete list of specifiers.

The localizer can reorder the arguments in the translated string if that is necessary. If a string contains multiple variable arguments, you can change the order of the arguments by using the *n*`$` modifier where *n* indicates the order of the argument. For example:

```
/* Message in alert dialog when something fails */
"%@ Error! %@ failed!" = "%2$@ blah blah, %1$@ blah!";
```

Just as in C, some characters must be prefixed with a backslash to be included in the string properly. These characters include double quotations marks, backslash, and carriage return. You can also specify carriage returns with `\n`:

```
"File \"%@\" cannot be opened" = " ... ";
"Type \"OK\" when done" = " ... ";
```

Strings can include arbitrary Unicode characters with `\U` followed by up to four hexadecimal digits denoting the Unicode character; for instance, space, which is hexadecimal 20, is represented as `\U0020`. This option is useful if strings must include Unicode characters that cannot be typed for some reason.

Strings files are best saved in Unicode format. This allows them to be encoding-independent, and simplifies the encoding to use when an application loads strings files. The TextEdit application can save in Unicode format. You can select the encoding either from the Save dialog in Plain Text mode, or as a general preference for TextEdit.

Generating Strings Files

Although you can create a strings files by hand, you can also generate one automatically from your source code using the command-line program `genstrings`.

The program works by parsing the source files that you specify, extracting the information from each call to Cocoa's `NSLocalizedString` macro (and variant) and Core Foundation's `CFCopyLocalizedString` macro (and variants), and adding that information to the appropriate strings file. Every entry generated from a call to one of the relevant macros is placed in a file called *table*`.strings` where *table* derives from the "table" argument of the macro (`Localizable.strings` by default if no table is specified). Using separate tables creates separate domains for sets of strings, allowing different translations of the same string depending on the context.

The comment provided in these calls is also written out to the strings file, allowing the translator to get a better idea of what the string is used for. It's important to understand that `genstrings` generates one entry for each call to one of the related macros and duplicates any identical entries. If your code has more than one of these macros with the same arguments, you'll have to edit the strings file after you run `genstrings` to remove the redundant entries. Although a key can occur multiple times in a source file, each key in a strings file must be unique. (However, you can have multiple strings files, or "tables," per localization, and each of these files can contain the same key.)

Whether you want to generate strings files automatically or create them by hand is up to you. In some cases you might find it convenient to generate your strings files once in the lifetime of your application development, then tweak them by hand. However, in most cases, it's better to generate new strings files from the source whenever you change or add any localized strings.

Adding Multiscript Support

An application that includes multiscript support can accurately display text in various scripts simultaneously. Such an application can accept textual input, display text, and print text containing the scripts of different languages in the same document, regardless of a user's language preferences. If the application is not prepared to offer multiscript support, some of this text probably appears garbled.

Multiscript support is becoming an increasingly important and expected feature not only for operating systems but for third-party applications. With an internationalized operating system such as Mac OS X, some users expect to be able to create a document in one language and script, change their language preference, and then open the document as they last saved it. In addition, the Internet is fostering more expectations for this support; users frequently download foreign-language text that is different from the user's primary language.

On a general level, you attain multiscript support on Mac OS X by using the appropriate Unicode technologies and APIs. Specifically, this entails the following for Carbon applications:

- When possible, use CFString from Core Foundation String Services instead of C or Pascal strings.

  CFString objects internally store and handle Unicode data without requiring you to have any specific knowledge of the global character-set standard. If you need to convert between Unicode and C and Pascal strings in other encodings, use the facilities that CFString provides. However, you should avoid converting CFString objects to and from C strings or Pascal strings as much as possible. Such conversions are not only costly but frequently introduce bugs affecting multiscript presentation. If you cannot find any CFString or Unicode-aware APIs to use in a certain situation, you should convert the encoding to the per-user default system encoding.

- For localized strings, use Core Foundation's `CFCopyLocalizedString` (and related macros) instead of `GetIndString`.

  `GetIndString` is based on the script system so it cannot represent multiscript text. `CFCopyLocalizedString` can represent such text because it returns a Unicode-based CFString object. Generally, your code should use dynamic text processing (such as is afforded by the localized-string mechanism described in "Localizing Strings" (page 217)) over static text in your code.

- Avoid directly accessing system layers below Core Services.

  You should be able to obtain most of the functionality available in the kernel environment (particularly BSD and Mach calls) by using the Core Services frameworks, Core Foundation, in particular. As much as possible, avoid calling BSD functions directly. For accessing the file system, use Core Foundation URL Services (CFURL), the File Manager data type `FSRef`, or (in Cocoa) NSFileManager.

- Avoid using the TextEdit API.

The TextEdit API is capable of dealing with multiscript text. However, it requires you to manage script fonts and style runs yourself. MLTE provides a much simpler API to handle multiscript text based on Unicode.

- Never assume text data to be in the MacRoman encoding.

  You can no longer assume that all text data is in MacRoman or ignore text encoding issues altogether. You must be prepared to handle text encoding issues. Untagged text data unaccompanied by script code is not necessarily in the system script (Roman) anymore. If this assumption is wrong, as it can often be, users are presented with garbled text. Worse, it could lead to anomalies that corrupt user data or even crash the system.

The Cocoa APIs, properly used, automatically provide multiscript support.

For more information on file encodings in Mac OS X, see "File Encodings and Fonts" (page 184) in the chapter "The File System" (page 163)."

CHAPTER 13

Installation and Integration

The moment arrives. You've worked for months on your Mac OS X application, designing, coding, redesigning, debugging, testing, tuning, and fixing bugs. Now, with one final build, your application should emerge as what it was meant to be: an elegant, rock-steady feat of engineering, purring with power and potential.

But wait. Are you sure it's ready? Is it truly in shape for deployment? Is it of commercial quality, and can customers easily install it and use it? Is there anything you have overlooked?

This chapter tries to help you answer these questions. First, it provides a checklist of important tasks that you should complete (or at least consider completing) before deploying your application. Some of these tasks are in the realm of fit-and-finish but others are essential to a well-designed application. Second, it discusses issues affecting how your application can best be integrated with the various pieces of Mac OS X as well as with other applications. Finally, this chapter describes the various approaches you can take and tools you can use for installing your application. You've worked hard to produce a great application with the feature set your customers value. Now take the time to ensure that they can enjoy your application by delivering a great installation and setup experience.

Preparing Software for Mac OS X

To software developers, Mac OS X is a very accommodating system. It often gives you alternative approaches for accomplishing the same goal, with one approach the same as (or close to) what has been done traditionally. Areas where such alternatives exist include application packaging, resource handling, and document typing. However, one approach is often better than another and sometimes you can combine approaches. The following section describes various important facets of application design, discussing not only what you can do, but more importantly what you *should* do for reasons of performance, interoperability, or robustness.

Applications and Documents FAQ

Information about applications and documents is scattered throughout this book because many different factors affect the nature, structure, and handling of applications and documents. These factors include bundles, executable formats, file systems, and the Finder. This section brings together this information in a question-answer format, summarizing the salient points about documents and applications for developers.

What metadata must I specify for an application?

For users to launch your (bundled) application, the Finder application must be able to detect that a folder is a bundle, and then it must be able to discover that the bundle is an application. To make this determination, the Finder first checks for one of two things:

- whether the bundle bit is set on the bundle folder
- whether the extension of the bundle is one of those reserved for bundles (including `.app`)

If the Finder determines the folder is a bundle, it reads the CFBundlePackageType key stored in the bundle's `Info.plist` file; if this key contains the value `'APPL'`, it knows that the bundle is an application. If this key is not specified in the file, it uses the bundle extension (`.app` in the case of applications) to determine the bundle type.

Because the bundle bit, as with other forms of HFS and HFS+ metadata, can easily be stripped in a networked environment involving multiple file systems, it is important that your application bundles always have the extension of `.app`. Project Builder automatically appends this extension when you build an application, but other IDEs might not. In no case should you remove the extension or encourage your users to. If the "unsightliness" of `.app` bothers you, don't worry; the Mac OS X Finder suppresses the display of the `.app` extension.

Although Apple does not set the bundle bit on its applications, you may set this attribute on the bundle folder of your application when you build it.

For more on this subject, see "The Finder and Bundles" (page 121) and "The Handling of Applications and Documents" (page 192).

Must I package my CFM executable in a bundle?

The short answer is "no, but you probably should." See "CFM Executables" (page 229) for the long answer.

How should I store application resources?

In Mac OS 9 and prior releases of the Mac OS, applications put their resources in the resource fork of the application executable. For Mac OS X, this is no longer the recommended approach. Instead, applications should have their resources in the data fork of a separate file in the application bundle.

The reason for this recommendation is the same reason behind having filename extensions as well as Finder metadata for document typing (see "Why even have extensions?" (page 228)). The HFS and HFS+ volume formats permit files with multiple forks, or data streams. However, anything not in the data fork of a file can easily be stripped away as the file travels between heterogeneous computer systems in a local area network, an intranet, or the Internet. The point is to make resources and all other forms of data persistent in an increasingly networked world.

Developers of Carbon applications have to consider other factors related to resources on Mac OS X, especially if those applications depend on the Code Fragment Manager (CFM). If the application is a single-file executable managed by CFM (that is, not a bundled CFM application), the resources should go in the executable's resource fork. By default, applications packaged as a single-file CFM executable have their resource forks opened when they are launched. Conversely, applications that are packaged as bundles have their localized data-fork resources opened by default when they are launched.

More possibilities for resources open up if a Carbon application is packaged in a bundle. Instead of combining a resource with other resources managed by the Resource Manager, you can put a resource of a certain type in its own file. For example, if your application uses a TIFF image, you could put the TIFF image data in the data fork of a file having an extension of `.tiff`. Then, using the proper bundle APIs, you can access the resource directly. There are advantages to putting each resource in its own file. Such an approach, for instance, makes it easier to "export" resources specified in XML property lists and makes it easier on localizers, since they don't have know how to use another tool.

Carbon applications, whether they are CFM-based or dyld-based, can always use Resource Manager–style resources. However, if you package your application in a bundle (as is recommended) you should put your resources in files in the `Resources` directory of the bundle and you should use only the data forks of these files. These files, which should have an extension of `.rsrc`, are treated as bundle resources just like any other file, and are easily internationalized. Although `.rsrc` files can have any base name, if you give them standard names and put them in the standard bundle locations for resources, the system bundle routines manage the resources automatically. It works like this:

- Put nonlocalized resources in a file named *executableName*`.rsrc` and place this file in the bundle location for such resources (that is, directly in the `Resources` directory).
- Put localized resources in files named `Localized.rsrc` and place these files in the appropriate bundle locations (that is, the `.lproj` directories) for localized resources.

When the application is launched, the system bundle routines automatically open these resources and make them available to the application.

To summarize, the following options are available for application resources:

- Each resource of a particular type goes in its own file, which has an extension appropriate to the type. This approach is appropriate for bundled applications of any application environment. The exception to this "one-per-file" model is localized strings, which are collected for each localization and put in a file conventionally named `Localized.strings`; see "Localizing Strings" (page 217) for more information.
- Bundled Carbon applications can put their Resource Manager–style resources in the data forks of files, each with an extension of `.rsrc`. These files can be placed in the bundle locations for nonlocalized and localized resources.
- Nonbundled Carbon applications must put their Resource Manager–style resources in the resource fork of the application executable.

  If you want the Finder to handle the application and its documents properly, you must specify the key-value pairs otherwise found in a bundle's information property list as a resource of type `'plst'`, ID 0.

For further information on this subject, see "Bundles and the Resource Manager" (page 128) and "Resource Forks" (page 181).

How do I indicate document types in Mac OS X?

In Mac OS X, you indicate the type of a document by specifying two things:

- Type and creator codes stored as attributes of a file (if it is created on an HFS or HFS+ volume)
- One or more file extensions relevant to the type (for example, `.html` and `.htm`)

Apple recommends that your applications make use of both forms of document typing. If your application owns a document, you can specify both type and creator codes and file extensions in the information property list (`Info.plist`) of the application project (see "Information Property Lists" (page 198)). Project Builder provides a means for entering this information: the Application Settings pane for a build target. Your application should enforce the setting of all valid types for its documents, particularly file extensions. See "How should my application save documents?" (page 229).

There is one final caveat with regard to extensions. Applications in general should be prepared to open documents that have extensions but no type and creator codes. This behavior is especially expected for common (and hence cross-platform) document types, such as image files, text files, and HTML files.

Can I treat my plug-ins as documents?

Plug-ins or any other loadable bundles are file packages, which the Finder presents as files. Applications can claim loadable bundles as documents just as they can files. So the advice given in "How do I indicate document types in Mac OS X?" (page 227) applies to them. Always include extensions for loadable bundles and, if applicable, the type (`'BNDL'`) and creator codes to be written to the bundle's `Info.plist` file. See "What metadata must I specify for an application?" (page 224) for the typing information to associate with all bundles.

How does the Finder handle documents?

The Finder uses both a file's type and creator codes and the file's extension to determine the document's type and owning application. When the Finder displays a file in one of its windows, it uses this information to find the appropriate icon to show for the document. When the Finder responds to a user action with a file—let's say the user double-clicks an icon to open a document—it uses the document type as a key to look up the application to use for that action. Depending on the specificity of the typing information (for example, there is an extension but no type or creator codes), the Finder might

- immediately open the document in an application
- display a dialog enabling the user to select an application
- open the document in one of the applications claiming that document type

If a file has neither type and creator codes nor extension, the Finder treats it as a non-document file; the file is displayed with a generic icon and double-clicking it does not open it in any application.

Keep in mind that applications can treat loadable bundles as documents. Double-clicking a bundle should cause the application, if it recognizes the bundle, to load it. As it does with any other document, the application specifies the bundle's extension, type and creator codes, role, and other information in its information property list. Before it can handle the loadable bundle as a document, the Finder must determine that it is a bundle.

For more information, see "The Handling of Applications and Documents" (page 192).

Why even have extensions?

Some Macintosh software developers react to file extensions with dismay. As a means for specifying document type and ownership, extensions seem primitive compared to the type and creator codes and the other rich metadata made possible by the multifork HFS and HFS+ volume formats. Using extensions seems to be a step backwards.

This is true, but only in a limited context. Macintosh users do not live anymore within a parochial Macintosh world. In the Internet age, documents frequently travel around a heterogeneous network, going, for instance, from a home Macintosh to a Linux network server to a Windows computer on a corporate local area network. Each computer on this path may have a different notion not only of what constitutes a document type but what constitutes a file. Many computer systems define a document's type solely by well-known extensions (such as `.jpg`, `.mp3`, and `.html`). They might not know what to do with an extensionless file and treat it as an unknown type. They would also ignore the HFS+ metadata—or worse, strip it out altogether, so that it is irretrievably lost.

How should my application save documents?

Your application should save its documents only as one of the types that it claims an Editor role for. When your application saves a document file, Apple recommends that it associate the proper filename extension with it, as well as any defined type and creator codes. A user can later change or remove the extension (and pay the consequences for doing so), but your application should always apply all valid forms of document typing, including extensions, when it saves its documents. (See "Why even have extensions?" (page 228) for the reasons why.)

When your application can save documents under more than one type, Apple recommends that it present those types in a pop-up menu in the Save dialog (with any "native" document type being preselected). Applications then would handle extensions in the following way:

- If users type no extension in the filename field, add it for them.
- If users type the wrong extension, remove it and add the correct one.
- If users type the correct extension, accept it.

Another possible approach is to display an "untitled" filename with the correct extension appended but only the base filename selected; an example might be "Untitled-1.txt" where "Untitled-1" is selected.

CFM Executables

As noted earlier, Mac OS X is a very accommodating operating system. It supports multiple file systems, multiple application environments, multiple programming models, multiple graphics-rendering libraries, and multiple network protocol stacks. It also supports multiple runtime environments and executable formats. Specifically, the following types of binaries execute on Mac OS X:

- Mach-O code modules managed by the dynamic link editor (dyld)
- PEF code fragments managed by the Code Fragment Manager (CFM)
- Java class files managed by the Java virtual machine

Of the three executable formats, the Mach-O is preeminent. It is the native format upon which the others ultimately depend. CFM and PEF technology, which are the preeminent library manager and executable format in Mac OS 9, are bridges to the Mach-O/dyld technology, much in the manner CFM-68K was a bridge to PowerPC. Saying that Mach-O and dyld are the native executable format and library manager in Mac OS X means that all system frameworks, even Carbon frameworks, are built as dyld-managed binaries in the Mach-O format. However, CFM is the traditional Macintosh library manager and PEF is the traditional executable format for code fragments, and so many Macintosh IDEs currently generate application executables for this runtime environment (which includes the Classic compatibility environment).

As the section "Library Managers and Executable Formats" (page 267) explains, there are good reasons for building applications for Mac OS X as Mach-O executables. Foremost among these reasons is performance. The CFM/PEF runtime environment is layered on top of the dyld/Mach-O runtime environment; thus, code that is CFM/PEF-based must go through an additional layer of software in order to execute.

However, there is nothing to prevent you from building applications as CFM-managed binaries. Such binaries run without problems on Mac OS X, including in the Classic application environment. Indeed, there might be occasions when you want your CFM application run in the Classic environment rather than in the Carbon environment; for example, your application depends on plug-ins that have not been fully ported to Carbon. For these occasions, the Finder's Info window presents an option that enables users to launch a selected CFM application in Classic. If you want to override this option and have the application always launch in Carbon (or always launch in Classic), you can specify the appropriate Launch Services key in your information property list; see "Launch Services Keys" (page 286) in the chapter "Software Configuration" for details.

If you choose to deploy your application as a CFM executable, you must decide whether to package it in an application bundle. When packaging is considered (and Java is excluded), there are three different types of applications that can run on Mac OS X. Table 13-1 shows the possible types.

Table 13-1 Types of applications supported in Mac OS X

Library type	Single file	In bundle
CFM/PEF	yes	yes
dyld/Mach-O	no	yes

Ideally, you should package a CFM executable in an application bundle. By doing so, your application gains all the benefits that accrue to such packaging, which are itemized in "An Application Is a Bundle" (page 131). A bundled CFM application is easily launched on both Mac OS X and Mac OS 9, but in different ways. On Mac OS X, the user double-clicks the opaque file package and the Finder launches the application. On Mac OS 9, the user opens the `.app` folder and double-clicks the next thing he sees: an alias to the CFM executable. Also keep in mind the recommendation given in "Should You Use CFM or dyld?" (page 270). Ideally—from a performance perspective—your application bundle should have both kinds of executables: a CFM-managed executable optimized to run on Mac OS 9 and a dyld-managed executable optimized to run on Mac OS X.

Currently, there is no developmental technology for creating bundled CFM applications; you must create the bundle by hand, following the information given in the chapters "Bundles" (page 115) and "Application Packaging" (page 131). Fortunately, a short cut is available. If you have access to Project Builder, you can use it to create an empty application and reuse the generated bundle for your CFM application. Even with this shortcut, there are certain things to keep in mind when creating your CFM application bundle:

- The bundle directory itself should have a type of `'APPL'` and *may* have the bundle bit set. It should also have an extension of `.app`.
- The CFM executable should go in a directory named `MacOSClassic` directly under the `Contents` directory.
- At the top level of the bundle directory, create an alias (of the same name) to the CFM executable. The following listing illustrates this:

```
MyApp.app/
    MyApp /* alias to Contents/MacOSClassic/MyApp */
    Contents/
        MacOSClassic/
            MyApp
        ...
```

- Create an XML property list in a file named `Info.plist`; in this information property list specify all necessary key-value pairs as described in "Information Property Lists" (page 198). Then put the file immediately under the `Contents` directory.

Note: You can use the Property List Editor application (`/Developer/Applications/PropertyListEditor`) to help you create the property list. If you create the property list with some other editor, and the text contains non-ASCII characters, make sure the editor can save the file in UTF-8 encoding. You might also want to use an existing application's `Info.plist` file as a template.

- Put the application's resources in the bundle, following the instructions in "How should I store application resources?" (page 225).

If you wish, you may deploy a CFM executable as a single-file application that stores its Resource Manager–style resources in its resource fork. If you do this, and you want the Finder to handle the application and its documents properly, you must specify the contents of your application's information property list as a resource of type `'plst'`, ID 0. If a single-file executable does not have a `'plst'` resource, it is considered to be an application to be run only in the Classic environment. Through a Finder Info-window option, you can force-launch a CFM application into the Classic environment.

User Interface Issues

You might have to consider several user interface issues before your application is ready for deployment. First and foremost, you should ensure your application conforms to what is said in *Inside Mac OS X: Aqua Human Interface Guidelines*. You can obtain a PDF version of this book from the section of the Apple Developer Connection website listing Mac OS X technical documentation at:

http://developer.apple.com/techpubs/macosx/

Second, make sure that your application is properly internationalized and localized for all languages and regions where it will be marketed; as part of internationalization, ensure that your application can support the presentation of multiple scripts in one document. For information on these topics, see the chapter "Internationalization" (page 209).

The following sections discuss other aspects of your application's user interface related to development.

Icons

An application or document icon must be an `'icns'` resource contained in the data fork of a file having an extension of `.icns`. Apple provides two applications (in `/Developer/Applications`) to help you create and manage icons:

- Icon Composer—Imports an image in most standard bitmap formats (including TIFF, PICT, JPEG, and GIF) and converts it to a set of icons of pixel dimensions 16 x 16, 32 x 32, 64 x 64, and 128 x 128. It also creates a bit mask for the first three sizes. For best results, you should make versions of an icon in each of the four sizes; also, the imported image should be of equal height and width. The application saves the icon data as a file with an extension of `.icns`.
- Pixie—Displays portions of the screen at various magnifications and allows you to copy those magnified images to the Clipboard or save them as TIFF files.

The Grab application in `/Applications/Utilities` can also be useful for icon composition since it can capture (as a TIFF file) the entire screen or a part of the screen.

Users can assign custom icons to documents just as they can in Mac OS 9. To do so, they must paste a copy of the custom icon into the well holding the current icon displayed in the Finder's Info window (File > Show Info). To enable users to do this, you must override the default document icon in the Finder metadata (Finder Info).

Custom Controls and System Appearance

If you create a custom control that draws itself, you must ensure that it visually conforms to the Aqua appearance the user has chosen in the General pane of the System Preferences application. Currently the appearance—a color that is applied to buttons, menus, and windows—is either Blue or Graphite.

To make a Carbon custom control comply with the system appearance, your custom control definition must take advantage of the Appearance Manager APIs (declared in `Appearance.h`). For a custom Cocoa control to be compatible with the selected Aqua appearance, the custom subclass of an Application Kit class should use the NSColor `colorForControlTint:` method to get the NSColor object corresponding to the current Aqua tint selection, and then use that object to colorize its own drawing.

Generally you should avoid creating custom controls that draw themselves because there is always the possibility of some incompatibility with Aqua. If you do create a custom control, the new behavior should ideally involve something other than drawing.

Carbon Nib Files

Carbon projects can use Interface Builder to create user interfaces from palettes of "objects" just as Cocoa projects can. The objects on the palettes include buttons, text fields, sliders, menu items, windows, and progress bars. You can set attributes, dimensions, and control information for these objects. Interface Builder saves interfaces as XML archives known as nib files since they have an extension of `.nib`.

A Carbon nib file can contain definitions for most of the user-interface objects typically found in Carbon applications. It also incorporates functionality for features such as the Carbon event model. Special Carbon APIs give your application access to the objects defined in the nib file. Nib files are not only a convenient way to store the specification of an interface, but can be used to localize versions of a user interface. And, because a nib file is XML-based, it is inherently more exportable to other environments.

For these reasons, Apple recommends that Carbon developers start converting many of the user-interface elements previously stored as Resource Manager–style resources to Interface Builder objects. See the project example in `/Developer/Examples/InterfaceBuilder/IBCarbonExample` to get an idea of what is possible. Note that Carbon nib files lack many of the capabilities defined in Cocoa nib files (target-action connection, for example) largely because of the differences between the procedural and object-oriented programming models.

Ownership and Permissions

At a fundamental level, Mac OS X is a BSD system. A part of this underpinning is the way BSD implements ownership of, and permissions for, files and folders in the file system. This model, in turn, controls who can read, write to, rename, and execute files, and who can copy and move files to and from folders. Although the model is conventionally associated with UFS or similar file systems, Mac OS X extends it to all supported file systems, including Mac OS Standard (HFS) and Mac OS Extended (HFS+).

Although a thorough description of ownership and permissions on traditional BSD systems is beyond the scope of this book, some understanding of it is necessary both to see how it differs from the permissions model in Mac OS 9 and how the implementation in Mac OS X differs from the traditional BSD model. So this section starts with a quick summary of the BSD permissions model before discussing these differences. (For a more complete description, consult one of the numerous books or websites devoted to BSD.)

Overview of BSD Permissions

For each folder and file in the file system, BSD has three categories of users: owner, group, and other. For each of these types of user, three specific permissions affect access to the file or folder: read, write, and execute. For example, if a user does not have execute permissions in any of the categories for an application, he or she cannot run that application.

- An **owner** of a folder or file is generally the user who created it. Owners typically have full privileges (read, write, and execute) for that file or folder; they can set the permissions for other classes of users and, if they are the root (explained below), can even transfer ownership.
- Every user is also in one or more **groups**. A group is a named collection of users that have something in common and to which group permissions apply. For example, you might have a group—for example, "projectx"—consisting of all users involved in an engineering project. The members of this hypothetical group are given write permissions for source files in a code repository. The group owner of a folder is the default group owner of any files created in that folder. The administrator of a system is responsible for setting up the appropriate groups.
- Users in the **other** category are just that—everyone other than the owner and group users—and their permissions for a file or folder are generally the most restrictive.

On BSD systems there is a special root user, also known as the superuser. Root users have unlimited access to the folders and files on the devices attached to the computer they are using; they can

- read, write, and execute any file
- copy, move, and rename any file or folder
- transfer ownership and reset permissions for any user

There is a special group called the wheel group. Membership in the wheel group confers on users the ability to become the superuser after entering `su` at the command line and providing a password.

In a BSD shell, if you enter the command `ls -l` for some location in the file system (say, `~steve/Documents`), you get results similar to the following:

```
total 3704
drwxrwxrwx  5 steve  staff  264 Oct 24 20:56 General
drwxrwxr-x  5 steve  admin  264 Oct 21 21:47 ProjectDocs
drwxr-xr-x  6 steve  staff  160 Oct 25 12:00 Planning
drwx--x--x  6 steve  staff  160 Oct 21 15:22 Private
-rwxrwxrwx  1 steve  staff    0 Oct 23 09:55 picture clipping
[spongebob:~/Documents] steve%
```

The results show, among other things, the owner and primary group for a file or folder and what the permissions are for each category of user. Users are shown in the third column and the primary group in the fourth; thus, for the `General` folder, the user is `steve` and the group is `staff`. The first column is a set of ten coded "bits" that represents the type of file-system entity and the permissions for that entity. An initial `d` indicates that the item is a folder (directory); if the initial position is a dash, the item is an ordinary file. The remaining nine bits fall into three implicit groups representing first owner, then group, then other. The `r`, `w`, and `x` characters, if present, indicate that read, write, and execute permissions are turned on for the type of user the set of bits applies to.

Look at the permissions for the `Planning` folder in `~steve/Documents` as an example:

```
drwxr-xr-x  6 steve  staff  160 Oct 25 12:00 Planning
```

The owner permissions for this folder are `rwx`, meaning that the owner of the folder (the holder of the `steve` account) can copy files and folders to this directory and can make it his or her current working directory (through the `cd` command) and execute any program in it. The permissions for anyone in the staff group and anyone else are both `r-x`, meaning that those users can read files in `steve` and can make it their current working directory, but they cannot write files to that folder or modify or delete files in that folder.

The `ProjectDocs` folder has different permissions:

```
drwxrwxr-x  5 steve  admin  264 Oct 21 21:47 ProjectDocs
```

Here read, execute, *and* write access are turned on for the group (`rwx`), but you have to be a member of group `admin` in order to be granted this access.

Each read, write, execute triplet can also be represented as a decimal number between 0 and 7. XREF lists the relationships between number and triplet.

Table 13-2 File permission mappings

Decimal Number	**Permission**	**English translation**
0	---	No permissions
1	--x	Execute only
2	-w-	Write only
3	-wx	Write and execute
4	r--	Read only
5	r-x	Read and execute
6	rw-	Read and write
7	rwx	Read, write, and execute

For example, look at the permissions for the `Private` folder.

```
drwx--x--x  6 steve  staff  160 Oct 21 15:22 Private
```

Here you can represent the triplet `rwx` with the number `7`, and the triplet `--x` with the number `1`. Thus, you can express the permissions for the `Private` folder as `711`. The numerical representation is useful with tools that change file permission.

If you have the appropriate permissions, you can change owner, group, and individual permissions from a Terminal shell using, respectively, the `chown`, `chgrp`, and `chmod` commands. See the associated man pages for details. You can also see the same ownership and permissions information for a selected file or folder in the Privileges pane in an Info window in the Finder. If you are the owner of the file or folder, you can also change permissions in this window.

File Permissions in Mac OS X

Perhaps the biggest difference between traditional BSD permissions semantics and the Mac OS X implementation is that the root user is disabled in Mac OS X after system installation. Even if users belong to the wheel group—the traditional way of granting superuser privileges—they cannot become the superuser via the `su` command. The reason for doing this is apparent: security. Because the root user has unlimited power over a file system, he or she can potentially wreak havoc, intentionally or not. Security is even more of a concern if logging in remotely or other forms of remote access are enabled.

However, Mac OS X provides the administrator user in place of the root user. The administrator can perform almost all functions the root user can, and can do them using the Finder (that is, without resorting to the command line). The only thing the administrator is prevented from doing is directly adding, modifying, or deleting files in the system domain; however, an administrator can use special applications such as Installer or Software Update for this purpose. The administrator is not a real user (in the sense of a user with an account of "admin"); an administrator is any user who belongs to the admin group.

The user who installs Mac OS X on a system and who provides information to the Setup Assistant application automatically becomes the first administrator for the system. Thereafter, this user (or any other administrator) can use the Users pane of Accounts System Preferences to create accounts on the local system for new users. An administrator can grant administrative privileges to a new user by setting the appropriate options on the user's account. Administrative users belong to the "admin" group. Non-administrative users belong to the "staff" group.

Although the root user is disabled by default, you can, if you're an administrator, reenable it and acquire superuser status. But, for security reasons, you should do this only if circumstances absolutely require it. To reenable root, run the NetInfo Manager application in `/Applications/Utilities` and authenticate yourself as the local administrator. Then choose Enable Root User from the Security menu; this menu item is enabled only if you are a member of the local admin group and you have been previously authenticated in the local domain. Once you're enabled as root user, your password is blank, so it is recommended that you give root a password (via the Domain > Security > Change Root Password command). After you've completed the task requiring root access, you should relinquish superuser privileges by choosing Disable Root User from the same menu.

Permissions for Applications and Documents

When you build an application with Project Builder, the build subsystem automatically sets the permissions of the executable file to `-rwxr-xr-x`; this setting enables the owner—that is, the person who installs the application—to execute and write to the application, whereas all others can only execute it. Other IDEs set similar permissions on built executables.

The `-rwxr-xr-x` setting should suffice except in the rare situations where an application requires privileged (root) access. An example would be an application such as a disk repairer that requires low-level hardware access through the kernel. In cases such as this, you should install the application `setuid` to acquire root access for the application; then use the features of the NetInfo Kit and System frameworks that allow you to authenticate administrators. For more information on `setuid`, consult the `setuid (2)` and `chmod (1)` man pages.

Although the permission set of the application (particularly the "x" bits) determines who can launch an application, once the application is launched, the process is owned by the user who launches it. This means the application has the same access rights as the logged-in user, given that person's owner and group identities. Consequently, if that user has permission to write a document to a certain location, the application is able to save a document there.

When a Carbon, Cocoa, or Java application saves a document, the respective application environment automatically sets the permissions of the document as determined by the user's `umask` value which, by default, prohibits write access for group and other. This common setting for documents (`-rw-r--r--`) allows the owner to read and write to the file whereas all others may only read the file. If you want different permissions for your application's documents, you must use a file-management API that lets you set permissions; for Carbon these are provided by the File Manager and for Cocoa by the NSFileManager class.

The Classic Environment and Your Application

The Classic compatibility environment (or simply, Classic) makes it possible for the latest version of Mac OS 9, and all the applications capable of running on that version, to run on a Mac OS X system. As you would expect, there are strong similarities between Mac OS 9 in Classic and a "native" Mac OS 9 system. However, there are also differences, and some of these differences are significant, especially to application developers.

This section describes the Classic environment, especially its compatibility with native Mac OS 9 systems and its integration with the rest of Mac OS X. It informs developers of Mac OS 9 applications about things they should take into consideration if those applications are run in Classic. It also tells developers of software for the application environments of Mac OS X—Carbon, Cocoa, and Java—about aspects of design that might cause problems for applications running in the Classic environment.

A brief note on terminology: Sometimes the following discussion explicitly compares the Classic environment to Mac OS X; what is implied, of course, is "the rest of Mac OS X."

Overview of the Classic Environment

The Classic environment is called a "software compatibility" environment because it enables applications built for earlier versions of the Mac OS to run on a Mac OS X system. Thus a user can still use legacy applications until a complete transition to Mac OS X occurs. This process will take place over time, so the Classic environment is a necessary component of the operating system for the near future.

To the Mac OS 9 operating system that it hosts, Classic appears as a new hardware platform. It implements hardware services using the Mac OS X kernel environment (particularly the I/O Kit). The Classic environment is not an emulator; Mac OS 9 runs natively in it. It is visually and functionally compatible with the rest of Mac OS X so that to users—with the exceptions noted in "Integration With Mac OS X" (page 242))—it is largely indistinguishable from the other environments of Mac OS X.

Because of architectural differences, Mac OS 9 and the applications running in the Classic environment do not share the full advantages of the kernel environment, including memory protection and preemptive multitasking. Thus, if an application running in the Classic environment crashes or hangs, the environment itself sometimes has to be restarted. But it is only the Classic environment that has to be restarted, not the Mac OS X system that acts as its host.

Compatibility With Native Mac OS 9

Programs that run on a native Mac OS 9 system will probably not run in the Classic environment if they attempt to do anything directly at the lower layers of the system. Specifically, this means a number of different things:

- Applications that draw to the Window Manager port generally have their drawing clipped to the part that is over Classic windows. Going full screen, however, is supported.
- Applications that draw directly to the frame buffer, bypassing QuickDraw, will probably draw over Aqua windows unless they follow the shield cursor/show cursor conventions.
- Software that touches memory-mapped I/O registers will crash.
- Software that patches traps and expects global effect will get that effect on Classic but not on Mac OS X, where it might cause unexpected behaviors. For example, software that patches all file system traps to intercept file I/O will only see I/O within Classic.
- Software that attempts to access the file system directly at the level of a disk driver or through Device Manager APIs will not work. Consequently, disk utilities that work on native Mac OS 9 do not work in the Classic environment.

Generally, programs that modify or rely on Mac OS internals below the hardware abstraction provided by the kernel environment (and especially the I/O Kit) will not work in the Classic environment. These programs should instead use a higher-level API, if one is available, for such access. For example, all file-system access should be through the File Manager API.

Device Support

Most devices that Mac OS X generally supports are also supported for the Classic environment (or are planned to be supported soon). The classes of supported devices include the following:

- USB
- sound (in and out)
- disk images and SMIs (Self Mounting Image files)
- Ethernet

- SCSI (forthcoming)
- FireWire (forthcoming)
- video (forthcoming)

Mac OS X mounts all block storage (“disk”) devices, and the Classic environment sees them as volumes through the File Manager API. The Classic environment can grab access to a device if Mac OS X hasn’t. Disk images (including SMIs) and AppleShare volumes mounted in Mac OS X appear through the File Manager API within Classic. Note that Mac OS X always grabs access to the USB keyboard and mouse; Classic communicates with these devices at a higher level of event.

Device types that Classic won’t support include the following:

- ADB (except for the primary keyboard and pointing device, but these are only supported at the event level)
- LocalTalk
- internal floppy disk
- serial
- specialized PCI/PC cards (that is, any cards other than those that Mac OS X supports for display, networking, or SCSI)

For PPP (modem) and AirPort communication, a PPP connection in Mac OS X is available as a network connection in Classic.

Integration With Mac OS X

Wherever possible, Apple has tried to make elements of the Classic environment indistinguishable from their counterparts in Mac OS X. However, not everything looks or behaves the same. Some of these differences will go away over time; others reflect semantic differences that are permanent.

This section discusses areas of integration where there are known differences. It is possible that other differences might be introduced in a later version of the operating system. As as rule of thumb, if you don’t know if a specific Classic element is integrated with the rest of Mac OS X, assume that it isn’t.

User Interface

The differences between windows and other user-interface elements in the Classic environment and the rest of Mac OS X are probably the most conspicuous. Instead of Aqua, windows have the platinum look and feel. For example, instead of the translucent red, yellow, and green window controls, Classic windows sport the close box, the collapse box, and the zoom box. When you click the collapse (or windowshade) box, there is no special “genie” effect that hides the window as in Aqua; the window’s content simply disappears. A platinum window casts no “shadow.”

You can also see several differences in the menu conventions adopted by Classic applications and other Mac OS X applications. In Mac OS X applications, for instance, the placement of the menu command for terminating an application is different; a Classic application has the Quit menu item in the File menu whereas a Mac OS X application has the “Quit *application*” menu item in the application menu.

The differences extend to actions such as resizing and dragging windows. In Aqua, the window is redrawn at each point. However, when you drag or resize a Classic window, you see an outline of the window (a marquee) and the window is not redrawn until the operation ends.

Classic doesn’t take part in transparency effects, and when a Classic object is below a semi-transparent Aqua window, the object might appear completely white. This is not the case when a Classic window is over an Aqua object.

Another obvious difference is the file-system browser. Mac OS X makes use only of Navigation Services for this feature whereas Classic applications can use Navigation Services or the Standard File Package.

On the integration side, both Classic and Mac OS X support OpenGL and 8-bit graphics.

The Classic Environment and File Systems

The Classic environment supports most of the file systems supported by Mac OS X, including Mac OS Standard (HFS), Mac OS Extended (HFS+), AFP, ISO 9660, and UDF. However, it does not support UFS or NFS file systems. Consequently, a Classic application cannot read from, write to, or even see an NFS or UFS volume.

The Classic environment also differs from Mac OS X in the manner in which it handles file-system access permissions. Although Mac OS 9 itself is not aware of permissions on local disk volumes, AFP recognizes permissions on a per-folder basis; access to files is determined by the permissions assigned to the containing folder. Classic maps BSD permissions failures to the closest corresponding AFP permissions error, which results in the most compatible behavior for applications running in the Classic environment.

Applications (and especially installers) in the Classic environment can encounter problems because of this permissions-integration model. Those that do not properly handle AFP permissions errors might present misleading error messages or even malfunction. Another potential problem related to permissions is when the same disk or partition is used for both Classic and Mac OS X. Because only the root owner can write to the root of this boot volume, any attempt to install software at the root of the file system will fail. The installers for such applications must prompt the user to select a folder.

Note: Often the only recourse for installers with such problems on Mac OS X is for the user to boot back into native Mac OS 9 and install there.

Classic respects BSD file permissions except in a few special places, such as the System Folder, Trash, the Desktop Folder, and certain hidden folders specific to Mac OS 9. If a user has read-write permissions at the root of the Classic environment's file system, a Classic application can write to the System Folder and can move items to the Trash folder. When a program running in the Classic environment attempts to write to a folder where it doesn't have permission, an AFP error code is returned.

For AFP file sharing, Classic goes through Mac OS X.

Extensions and Preferences

In a typical setup of a Macintosh computer system, a native Mac OS 9 system is first installed on a partition of a hard disk or on a separate hard disk. Then Mac OS X is installed on the system and the Mac OS 9 volume is chosen as the Classic environment's start-up disk. With this setup, the user can use the same set of applications, extensions, preferences, and other software in both the Classic environment and the native Mac OS 9 system.

There can be conflicts between the set of extensions used in a native Mac OS 9 system and the corresponding extensions in the Classic environment. These conflicts can lead to crashes. Generally, when there are conflicting extensions, Classic disables the corresponding "native" extension when it starts up. A case in point is the Multiple Users extension. Mac OS X is inherently a multiple-user system. Therefore, the Classic environment disables the Multiple Users extension when it starts up; when the user boots into the native Mac OS 9 system, the Multiple Users extension is reenabled.

Note: The release notes for the Classic environment list the currently known set of conflicting extensions.

The Classic environment handles conflicting preferences (generally related to networking) in a similar manner. Any preference the user sets using the System Preferences application in Mac OS X overrides any corresponding preference set for the native Mac OS 9 system. However, when the user boots into the native Mac OS 9 system, all "native" preferences are restored. Classic does not make any permanent changes that will adversely affect the configuration of the native Mac OS 9 system.

The Finder and the Desktop

The Classic environment shares the Mac OS X desktop with all other application environments. With the Mac OS X Finder, users can manipulate files and applications in all environments, including Classic. With the Finder, they can

- navigate through the file system
- move, copy, delete, and rename files and folders
- launch applications by double-clicking them
- open documents by double-clicking them

The Classic environment presents a composite desktop in a manner similar to native Mac OS 9. This composite desktop includes the Mac OS X Desktop folder for the boot volume and the desktop folders from all other mounted volumes that Classic can see (thus excluding UFS or NFS volumes). Classic treats AFP volumes in the same way that native Mac OS 9 treats them. If such a volume has a Desktop Folder, it is excluded from the composite desktop but accessible at the root of the AFP volume through Classic's Standard File Package and Navigation Services APIs.

The Mac OS X Finder and Navigation Services hide the Mac OS 9 Desktop Folder if it is on the boot volume. In the scenario where Mac OS X is installed "over" Mac OS 9, if users want to get to the Mac OS 9 Desktop Folder from the Finder or Navigation Services, they can navigate there through an alias in `/MacOS9`. When users boot back to native Mac OS 9, they will see the Mac OS 9 desktop—the union of all non-AFP desktop folders on all mounted volumes—as they left it.

The Mac OS X Finder performs System Folder autorouting and Mac OS 9 System Folder blessing in the Classic environment.

Networking and Printing

Networking in the Classic environment is largely integrated with the networking facilities of Mac OS X. Classic shares with Mac OS X the 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.

In addition, the version of Internet Config used in the Classic environment is different from the Internet Config software used in Mac OS X. Calls to Internet Config made in Mac OS X look in the Mac OS X Internet Config database and calls made in Classic look in the Mac OS 9 Internet Config database. This can cause some confusion, since a request to launch your default browser can have different results depending on whether the request is resolved with Classic or native Mac OS X Internet Config and whether the configurations are different in the two databases.

Printer setup in the Classic environment is just as it is in a native Mac OS 9 system: Chooser or Desktop Printer Utility. The difference in Classic, however, is that the actual desktop printers are not shown on the desktop by the Mac OS X Finder. When printing from an application in the Classic environment, you can choose among configured printers in the Print dialog using the same pop-up menu as you would see in native Mac OS 9. When Classic prints, it falls back to using the Print Monitor rather than the Desktop Print Monitor.

Other Classic Integration Issues

There are a few other areas of Classic integration that are of interest to developers:

- **Fonts**. The Apple Type Solution (ATS) system integrates and manages fonts put into 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.
- **Copy/paste**. The Classic environment and the Cocoa application environment have some dissimilar data types used in Clipboard operations. Consequently some copy/paste or drag operations between these environments might fail.

Installing Your Application

With Mac OS X there are several options for installing your application. You can simply instruct users to drag the application to their hard disks, or you can prepare the application for an installer. This section describes these possibilities and summarizes the work required on your part to make an application installable.

Where to Install

As described in "How the File System Is Organized" (page 163), the standard directory layout of the file system has several places where applications can be installed:

- the combined system and local domains (`/Applications`)
- the user domain (`~/Applications`, either local or on the network)
- the network domain (for example, `/Network/Applications`)

Typically, most applications in a Mac OS X system are installed in `/Applications` to make them available to all users of a particular computer system. However, if the use of an application is to be limited—for example, it is purchased under the terms of a single-user license—it should be installed in the `Applications` folder in the user's home directory (this folder might first have to be created). If an application is to be executable by all users on a local area network, the administrator of that network should install the application on a file server accessible to those users.

Important
Do not install any software or resources, such as frameworks or fonts, anywhere in `/System/Library`. Such items should go in the appropriate locations in the local domain (`/Library`).

Users are not required to install applications in one of the domain locations. Because applications when packaged in bundles are designed to be largely self-contained, users can install them anywhere and they should execute without problems. However, by being outside the recommended locations, an application might not be able to take advantage of some system features, such as application services for Cocoa applications. Another missing feature would be Finder awareness of the application. The Finder, when it's building its application database, looks for applications in the known domains. If an application is outside all domains, that could affect the efficiency of the system. For more information on the Finder and applications, see the chapter "The Finder" (page 187).

Manual Installation

Because most applications for Mac OS X are packaged as self-contained bundles, all users need to do to install an application in most situations is to drag the bundle to a folder for which they have write permission. When you have a simple application where this type of installation is all that is required, you can provide instructions along with the application (in the form of a brief online or printed document) that tells users what to do.

The drag-and-drop type of installation is the preferred method on Mac OS X. Because an application keeps everything it needs within its bundle, simple manual installation reduces file-system clutter and eliminates dependencies on items residing elsewhere in the file system. It also gives users the option of not copying items to their hard disks that they are not particularly interested in (such as Read Me files). To uninstall an application, all users have to do is remove the application bundle from the volume.

The Finder supports a number of "application packaging" keys as a mechanism for simple installations of application components. When you build an application, you can specify these keys and their corresponding values in the application's information property list (`Info.plist`). When users then select the application and open the Finder's Info window, the Application Files pane of that window list the components. Users can select which components to install (or which to uninstall).

See "CFBundleVersion" (page 281) in the chapter "Software Configuration" for more information about these keys.

Installers

For some applications, simple drag-and-drop installation will not suffice. Either the requirements for preparing the application for execution are too complex, or the advantages of using an installer are too compelling. Among these advantages are

- compression
- displaying a Read Me file as part of the installation process
- requiring the user to agree to a license before installing
- requiring the user to authenticate as an administrator prior to installing
- a more professional-looking presentation

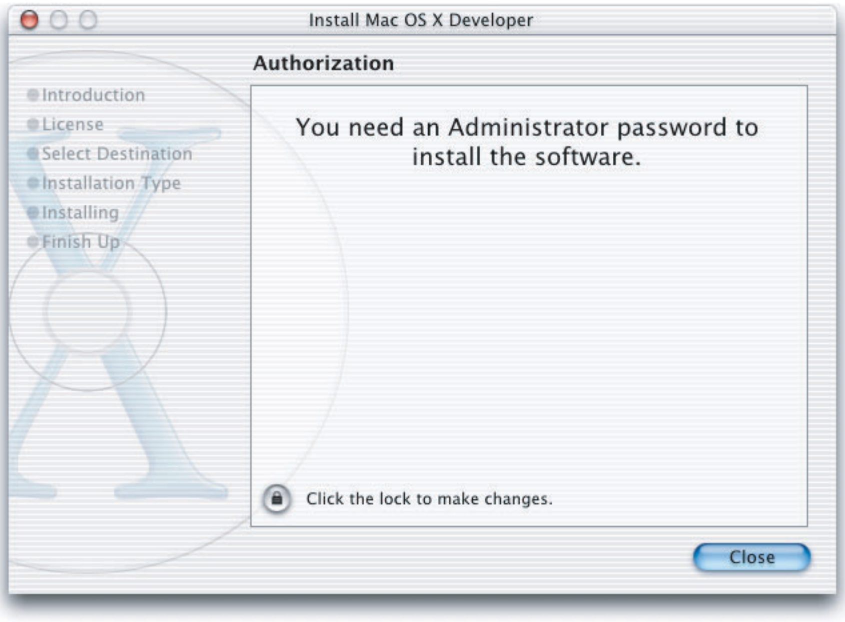
Important
Although an installer might have some advantages, Apple recommends the simple drag-and-drop manual installation whenever possible. You should use an installer only for applications that, for whatever reasons, need to install items outside their bundles.

If you decide to install your application with an installer, you have several options. If your application is a Cocoa or Java application—or a Carbon application that will only be installed in Mac OS X—you can use Mac OS X's native installer technology. If your application is a Carbon application, and you want to install this application on both Mac OS 9 and Mac OS X, you can take one of two approaches:

- For both platforms, use a third-party installer that has been ported to Mac OS X.
- Use the native installer technology for installing the application on Mac OS X and a third-party installer for Mac OS 9.

The native installer application for Mac OS X is called Installer, and it is located in `/Applications/Utilities`. When users double-click an installation package, the Installer application launches. The application steps the user through the installation process by presenting a series of panes for things such as authenticating the user's privileges (shown in Figure 13-1 (page 250)), specifying an installation location, and customizing the installation. The Installer application can also display the bill of materials and an installation log. The section "Installation Packages" (page 250) describes the structure and contents of an installation package.

Figure 13-1 Installer application



You need an Administrator password to install the software.

Click the lock to make changes.

You create an installable package for Mac OS X using another application named Package Maker (`/Developer/Applications/PackageMaker`). The general approach to preparing software for packaging and using Package Maker is described in "Creating an Installation Package" (page 251).

Installation Packages

The native form of installer-ready software on Mac OS X is called a package. An installation package is a file package—that is, a folder presented to the user as a file—that has an extension of `.pkg`. An installation package contains several components, most significantly the file archive, the Read Me and licensing documents (optional), and any installer scripts (optional). The format of installation packages enables Installer to provide useful features such as descriptive information, a default installation location, and other information.

Although the Finder presents a package as a single file-system entity, you can enter a package directory using the BSD `cd` command (via the Terminal application) and view the components of the package. The files that make up a package are named with suffixes that indicate the type of information stored in the file. A basic package contains these files:

- A bill of materials (`.bom`), a file in binary format that describes the contents of the package.
- An information file (`.info`), a text file that contains the information entered into the Package Maker application when the package was created.
- An archive file (`.pax`), an archive of the complete set of files to be installed. If the archive is compressed, the archive has an additional suffix of `.gz`.
- A size calculation file (`.sizes`), a text file that contains the compressed (and uncompressed) size of the unarchived software, which enables the installer to calculate space needed for installation.
- Additional (and optional) resources, such as icons, Read Me files, licensing information, pre- or post-install scripts, and so forth. These are package resources, used (or run) during installation but not installed along with the software.

Because installation packages are essentially folders (that is, file packages), their contents are susceptible to alteration and even corruption during handling. To prepare a package for distribution, you should archive and perhaps compress the package into a form that can be distributed safely.

Creating an Installation Package

The general procedure for creating an installation package involves two to three steps:

1. Create a distribution directory and copy the software to be installed to it.
2. Optionally, create an installation resources directory and put in it things such as Read Me files, licensing agreements, and install scripts.
3. Launch Package Maker and complete the "form" it presents to you.

See the following sections—"Create a Distribution Directory" (page 252), "Create a Resources Directory for Installation" (page 252), and "Complete the Package Maker Form" (page 253)—for information on each of these steps.

Create a Distribution Directory

Create a folder with a structure of subfolders that comprises a local "mirror" of the default directory hierarchy into which your software is to be installed. Then copy your software to the appropriate location (or locations). The directory hierarchy is "rooted" at a folder often given the name `dstroot` because it points at the distribution root (`/`). The contents of the distribution directory are made into the package's file archive.

For example, if you have built a framework and you want it installed in `/Library/Frameworks`, you might create the directory hierarchy `~/dstroot/Library/Frameworks` and copy the framework into this location. When Package Maker builds your package, it records this directory hierarchy. When your package is installed, Installer will place the software at the correct point in the file system. (However, if your package is relocatable, the user may choose to install it elsewhere.)

Create a Resources Directory for Installation

As described in "Installation Packages" (page 250), packages may contain supplementary resources used by the package during installation but which are not installed along with the software. Two common such resources are a Read Me file and a license document. The Read Me file contains information that users should (or might want to) read before they install the software. The license specifies the terms that the user must accept in order to use the software.

These Read Me and license files can be in one of three forms (with the appropriate extension): text (`.txt`), HTML (`.html`) or Rich Text Format (`.rtf`). When you create the files, use a text editor that can save the files in Unicode encoding. Special base names are reserved for the Read Me and license files. If the Read Me file is named `ReadMe.`*ext* and the license file is named `License.`*ext*, Installer recognizes each file (if it is present in a package) and handles it automatically:

- Installer adds a "Read Me" item to the bulleted list presented in the left column of the application. When the user reaches the Important Information pane, Installer displays the contents of the file. The user dismisses the Read Me by clicking the Continue button.

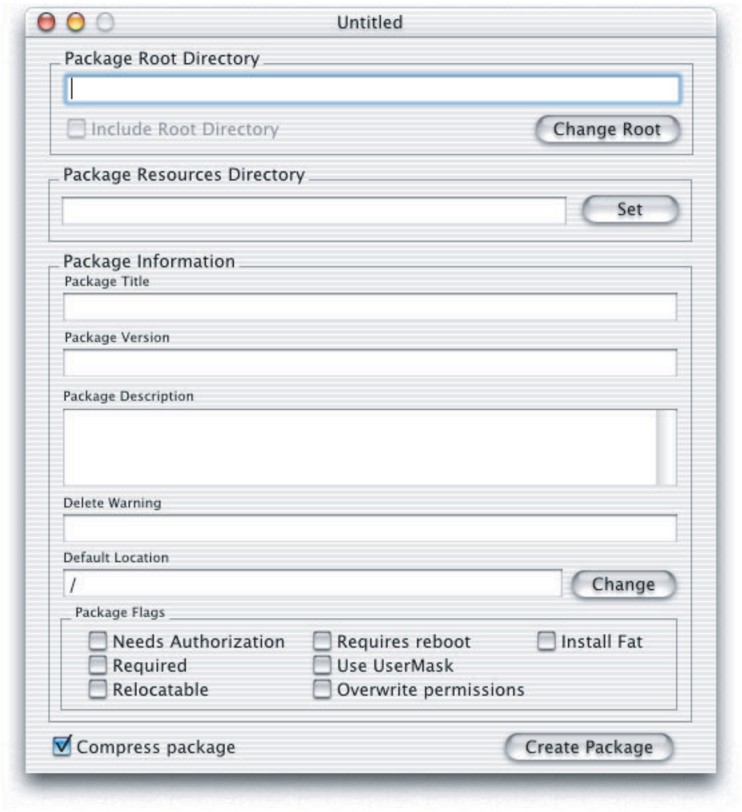
- Installer adds a "License" item to the bulleted list presented in the left column of the application. When the user reaches the Software License Agreement pane, Installer displays the contents of the file. When the user clicks the Continue button, Installer displays a dialog requesting that the user agree to the terms of license (by clicking Agree).

Because the supplementary installation resources are not installed along with your software, you should not place them in the distribution directory. Instead, create a "resources" directory at the same level as `dstroot` and put the files in there. The contents of the resources directory are copied into the package when it is created.

Complete the Package Maker Form

You use the Package Maker application to configure and create an installation package. It is easy to use; you just need to fill out the simple form of fields and checkboxes shown in Figure 13-2.

Figure 13-2 The Package Maker user interface



Untitled

Package Root Directory

[ ] Include Root Directory

Change Root

Package Resources Directory

Set

Package Information

Package Title

Package Version

Package Description

Delete Warning

Default Location

/

Change

Package Flags

[ ] Needs Authorization
[ ] Required
[ ] Relocatable
[ ] Requires reboot
[ ] Use UserMask
[ ] Overwrite permissions
[ ] Install Fat

[✓] Compress package

Create Package

Notice the first two fields: Package Root Directory and Package Resources Directory. These fields should contain paths that point to the roots of the distribution directory and the resources directory that you created earlier. See the application's online help for the purposes and expected values for these fields and for other fields and controls.

System-Wide Resources

If a bundled executable has resources that, for one reason or another, must be installed outside the bundle, the installer should take care of that. Often such resources need to be in certain locations to take advantage of system services. An example is fonts. The Apple Type Solution (ATS) system manages fonts by looking for them in `Library/Fonts` in the system, local, network, and user file-system domains.

Another issue related to file-system domains is the restriction of resources. For example, if you want a license-restricted font available to a single user rather than all users of a Mac OS X system, the installer should put it in the user's `~/Library/Fonts` folder instead of the `/Library/Fonts` folder.

You should be aware of the recommended domain locations for system-wide resources and have the installer put these resources in the proper location or in a user-selectable location. (See "How the File System Is Organized" (page 163) for information on these locations.) If the location requires system-administrator privileges, the installer should authenticate the user. The Mac OS X installer technology permits the installation of optional subpackages or subpackages that are to be installed in a location outside the application bundle. It uses the notion of metapackages to manage these collections of subpackages. Metapackages are file packages with a file extension of `.mpkg`. The installer presents a Customize button for each installable package that is based on a metapackage.

C H A P T E R 1 4

Issues and Options With Multiple Environments

Because Mac OS X is a highly layered system, there are often equivalent mechanisms at different layers. For example, threading APIs are available in Mach, in BSD, and in each of the application environments (because the latter are layered on top of the former). This section discusses some programming issues that might arise when there are different technologies and APIs (and even different terminologies) at each layer of the system.

Tasks and Processes

Different components make up Mac OS X, each with its own background, and this sometimes leads to clashes in terminology. This different terminology is often reflected in APIs as well as in documentation. The notions of "task" and "process" provide an important case in point. You have Mach tasks and BSD processes and Carbon Process Manager (CPM) processes and Multiprocessing Services tasks and so on.

A valuable aid for disambiguating this tangle is the following "equation":

Mach task = BSD process = Carbon Process Manager process

A Mach task, as defined by the Open Software Foundation, is a "container that holds a set of threads. More importantly, it contains those elements that the . . . threads need to execute, namely a port name space and a virtual address space." (*Mach 3 Kernel Principles*). In other words, the job of a Mach task (or BSD or Carbon Process Manager process) in Mac OS X is to manage memory, address spaces, and other resources related to the execution of its threads. Each Mach task (or process) has its own 4 gigabytes of virtual address space and this space is protected.

One Carbon Process Manager (CPM) process is layered on top of one BSD process. This layering enables the CPM APIs. Every Carbon, Cocoa, and Java application process is thus, at the same time, a Mach task, a BSD process (with its own process ID), and a CPM process (with its own PSN, or process serial number). Classic processes are an exception to the one-to-one process model. The applications running in Classic each have their own CPM process, but these multiple processes are layered on one BSD process.

Both Carbon and Cocoa include the name "task" elsewhere in their APIs. Cocoa uses "task" and "process" in the Mach and BSD senses. An object created with the Foundation framework's NSTask class is actually associated with a subprocess spun off from a parent process; it is a separate executable entity with its own set of threads and address space. Multiprocessing Services calls its user-level preemptive threads "tasks," largely to avoid conflict with the Thread Manager's (cooperative) threads. See the following section, ""Threading Packages" (page 258)," for more on Multiprocessing Services tasks.

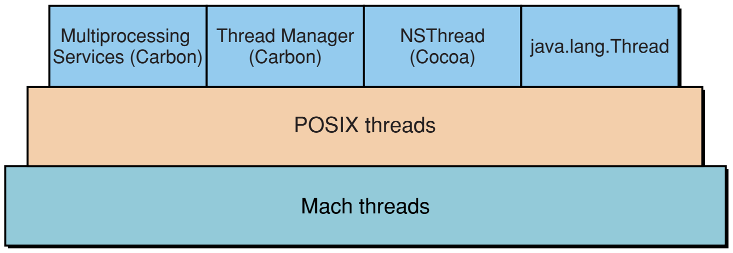
Threading Packages

A thread is an execution context within a process (see "Tasks and Processes" (page 257)). It is associated with a call stack and a processor's state. A thread shares virtual address space and other task-wide resources with other threads of the process. Threads are scheduled to run preemptively or, with symmetric multiprocessing, concurrently. Threading models built on top of the kernel's can, however, use various synchronization mechanisms to present cooperative threading behavior.

The capability for a process, such as an application, to have multiple executing threads is extremely valuable because it can enable greater program efficiency and simplifies the programming of some tasks. But multithreaded programming can also make some things more complicated.

Mac OS X gives developers a variety of models and programming interfaces for multithreading their programs. These packages have dependencies among themselves, since some packages are layered on top of others. Figure 14-1 depicts these packages and the dependencies.

Figure 14-1 Threading packages in Mac OS X



The kernel environment of Mac OS X, specifically Mach, provides the fundamental thread support. Mach maintains the register state of its threads and schedules them preemptively in relation to one another. In the case of symmetric multiprocessing, the kernel can preemptively schedule threads concurrently, one on each processor. The client API for Mach threads is implemented in the System framework.

The other threading models or packages are implemented on top of Mach threads.

Threading package	Description
POSIX threads	The thread package included with the kernel environment for implementing preemptively scheduled threads. It is one of the standard threading models in the industry. It is included in the System framework.
Multiprocessing Services	Package for preemptively scheduled threads on Carbon. It is layered on top of POSIX threads and is part of the Core Services layer.
Thread Manager	Package for cooperatively scheduled threads on Carbon. It is layered on top of POSIX threads and is part of the Core Services layer.
NSThread	Class whose objects wrap preemptively scheduled threads for use in Cocoa applications. It is layered on top of POSIX threads and is provided by the Foundation framework.
java.lang.Thread	Class whose objects wrap preemptively scheduled threads for use in Java applications. It is layered on top of POSIX threads.

You should always use one of the client APIs instead of the Mach APIs if possible.

Layering Details

As Figure 14-1 (page 259) illustrates, the BSD POSIX threads package (also known as Pthreads) layers its own multithreading environment on top of the kernel environment's Mach threads. The package schedules its threads preemptively and maintains a one-to-one mapping between a Mach thread and a POSIX thread.

The thread packages of the application environments are layered on top of POSIX threads. As with POSIX threads, they build their own multithreading environments on the threading substrata. The threads provided by Multiprocessing Services in Carbon and the NSThread class in Cocoa are preemptively scheduled and have a one-to-one mapping with the underlying POSIX thread. (In fact, the Multiprocessing Services threads, called "tasks" in the API, are thin covers for POSIX threads.) The Thread Manager's threads, on the other hand, are "multiplexed" onto a single POSIX thread and can be scheduled only cooperatively.

Usage Guidelines

When an application process is launched it automatically acquires one thread, regardless of the application environment. If you want your application to be multithreaded, you should use, in most cases, the thread package appropriate to your application environment and, for Carbon, to the type of required thread (preemptive or cooperative).

You should use POSIX threads when you want maximum source code compatibility with other operating systems. For example, a good deal of BSD code uses POSIX threads, which should be compatible with the implementation in Mac OS X.

With rare exceptions (such as debuggers), your projects should avoid creating and managing Mach threads. These threads lack much of the infrastructure provided by POSIX threads. Moreover, use of Mach threads is likely to lead to compatibility problems later.

Interprocess Communication

Usually at some point in its life, a process needs to communicate in some way with another process. Perhaps it needs to transfer some data, or it needs to let other processes know that something happened to it. However, communication between processes on a modern operating system can be a tricky affair; if such communication is poorly conceived and carried out, the overall stability and performance of the system could suffer.

In Mac OS X, a program has a number of ways to communicate with other programs. These mechanisms for interprocess communication often exists in different layers of the system. Each often has its own specific purposes, limitations, and intended scenarios. Some are more suitable than others for code written at a certain level of the system; for example, a kernel extension would not make use of Apple events.

The following mechanisms for interprocess communication exist in Mac OS X:

- Applications should prefer Apple events over all other methods for most interprocess communication on Mac OS X. See "Communicating With Apple Events" (page 262).
- Distributed notifications can be used to broadcast simple notifications to all applications on the same machine. See "Broadcasting Simple Notifications" (page 263).
- Consider using Core Foundation's CFMessagePort mechanism instead of Apple events in situations where performance is critical. See "Transferring Raw Data With CFMessagePort" (page 263).
- Use BSD sockets for communication over the network. See "Communicating With BSD Sockets" (page 263).
- BSD pipes can be used for atomic one-way communication on the same computer. See "Communicating With BSD Pipes" (page 264).
- BSD signals are invoked by the kernel to communicate exceptions to a process. See "Handling Exceptions With BSD Signals" (page 265).

- Combine shared memory with POSIX semaphores to share large resources such as pictures, sounds, or movies with other processes. See "Sharing Large Resources With Shared Memory" (page 265).
- NSPasteboard is a Cocoa class that allows simple runtime-persistent storage of publicly sharable data. Low-level interapplication Clipboard operations (cut, copy, paste) are implemented using NSPasteboard.
- Cocoa applications can use a services facility that allows them to advertise the services (through the Services menu) they can perform on behalf of other applications. See "Making Services Available to Other Applications" (page 266).
- Cocoa applications can also use distributed objects to send messages to objects residing in other threads or processes on the same computer. See "Calling Other Processes With Distributed Objects" (page 266).
- The Mach port object is the underlying primitive used for all interprocess communication on Mac OS X. See "Messaging With the Mach Port Object" (page 266).

Communicating With 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 method for interapplication communication on Mac OS X.

Apple event objects have a well-defined data structure with support for extensible, hierarchical data types. Applications typically use Apple events to request services and information from other applications, or to provide services and information in response to such requests. You can define your own custom events to suit your needs, but, to increase interoperability with other applications, it's a good idea to make the effort to adopt the standard set of Apple events documented by Apple.

In addition to low-level document management tasks such as "save document" and "open document", standard Apple event suites and related data structures are defined for many functional areas, including text handling and database management. A well-defined Apple event suite can also support a rich scripting interface through AppleScript.

Apple event objects can take a significant amount of time to create, so you will not usually want to use Apple events in performance-critical situations. To improve AppleEvent creation performance, try using the Carbon `AEBuild` and `AEStream` utilities, which are often significantly faster than `AEPutDesc` and `AEPutParam`.

Broadcasting Simple Notifications

A distributed notification is a message posted by any process to a per-machine notification center, which in turn broadcasts the message to any processes interested in receiving it. Included with the notification is an identifier of the sender, and, optionally, a 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 critical hardware, such as the network interface, or a typesetting machine.

There is no way to restrict the set of processes that are allowed to receive a distributed notification. Any process which registers for a given notification may receive it. Because distributed notifications use a string for the unique registration key, there exists potential for namespace conflicts.

Distributed notifications are true notifications because there is no opportunity for the receiver to reply to a notification.

Transferring Raw Data With CFMessagePort

The CoreFoundation CFMessagePort object implements a fast, efficient mechanism for transferring raw data from one process to another process on the same machine. If Apple events are too slow for your purpose, consider using CFMessagePort.

Communicating With BSD Sockets

Sockets support two-way communication between any number of processes. A socket is an object which associates an address with a file descriptor. Sockets should be used for all network communication on Mac OS X.

There are two primary variants of sockets, file and network. File sockets are addressed as filenames and for various reasons do not support communication between processes on different machines. Network sockets are addressed using the network host name combined with the port number (for example, `www.apple.com` and `80`). Both types of sockets are read and written using the POSIX calls `read` and `write`,

The Core Foundation framework includes an abstraction for sockets (CFSocket/CFRunLoop). Using CFSocket instead of raw sockets API calls allows Core Foundation to abstract any potential differences between operating systems, as well as providing a more object-oriented interface which you may find easier to program.

Using CFSocket with CFRunLoop allows you 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 is good for performance. CFMessagePort can also work with CFRunLoop.

Communicating With BSD Pipes

A pipe is a communication portal with one sending and one receiving end. Data written to a pipe is read in first-in, first-out (FIFO) order. In order to read or write data from a pipe, both the reading and the writing end of the pipe must be open.

Unnamed pipes must be created by a common ancestor process which then hands the pipe descriptor number to both child processes. This facility is typically used by the command line shell to connect processes which have been piped together (for example, "`cat magic.txt | grep -e Gwendoyln`" sends the contents of `magic.text` to the `grep` command via the C library console input stream).

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 receiving processes.

Reading and writing small amounts of data to a pipe can occur atomically if the size of the data written is below a certain, kernel-specified size. This allows the receiving end of the named pipe to avoid reading a partial buffer.

Handling Exceptions With BSD Signals

Signals are software interrupts that can be invoked on a specified process. The default signal handling behavior (provided by the system) usually terminates the process immediately on receipt of a signal. A process can override this behavior by installing a **signal handler** routine.

The most typical use of signals is by the kernel, which uses signals to notify a process of exceptional conditions such as invalid address errors and divide-by-zero errors. Another typical use is the command-line `kill` tool, which is capable of sending any user-specified signal to a process, though the most common use is to terminate a process with `SIGHUP`.

Signals are complex to use effectively, and they tend to behave differently (sometimes unreliably) on different operating systems. The signal namespace, being composed of a single integer, is limited, and collisions with either third-party signal numbers or numbers provided by future versions of the operating system are possible. As such, signals should generally be avoided for normal interprocess communication needs.

Sharing Large Resources With 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. The region is mapped into the address space of each process with access to it. Access to this region of memory is controlled through POSIX semaphores, which implement a kind of locking mechanism.

Shared memory has two distinct advantages over other forms of interprocess communication:

- Any process with appropriate permissions may read or write a shared memory region.
- The data is never copied, because any process can directly read it.

The disadvantage of shared memory is that it is very fragile. When a data structure in a shared memory region is corrupted, all processes which reference that data structure are also corrupted. For this reason, shared memory is best used simply as a repository for raw storage of data (such as raw pixels or audio), with the controlling data structures accessed through more conventional interprocess communication.

Making Services Available to Other Applications

The "standard services" facility lets a Cocoa application offer its functionality to other applications, without requiring the other applications to know in advance what's offered. A service-providing application advertises an operation that it can perform on a particular type of data—for example, encrypting the selected file or opening a selected URL. Any application that uses the services facility then automatically has access to that functionality through its Services menu. It doesn't need to know what the operations are in advance; it merely indicates what types of data it has, and the Services menu makes available the operations that apply to those types. The services facility thus gives Cocoa applications an open-ended means of extending each others' functionality. Transfer of data takes place through the Clipboard and can be either one-way or two-way.

Calling Other Processes With Distributed Objects

The Objective-C language runtime supports an interprocess messaging solution called "distributed objects". This mechanism enables a Cocoa application to call an object in a different Cocoa application. Calls may be *synchronous*, meaning the sending process is blocked while waiting for a reply from the receiver, or *asynchronous*, meaning no reply is expected and the sender is not blocked.

For more information on distributed objects, see *Inside Cocoa: Object-Oriented Programming and the Objective-C Language* and the *Foundation Framework Reference*.

Messaging With the Mach Port Object

Mach port objects implement a standard, safe, and efficient construct for transferring messages between processes. All other interprocess communications primitives on Mac OS X use the Mach port object at some level.

Because the sending and receiving process are scheduled independently of each other, there is no guarantee that a given process will be free to receive a message sent to it at any given time. Therefore, arriving messages are placed in a queue and retrieved at the convenience of the receiving process.

Processes may not access a given port without appropriate access permissions (or "port rights" in Mach terminology). The inherent stability of the Mach kernel is partly attributable to this mechanism.

You should not use Mach messaging directly if other alternatives are available, because the interfaces may change in future versions of the kernel.

The Mach port object is not related to and should not be confused with the internet address port number as used in BSD sockets (see "Communicating With BSD Sockets" (page 263)). The Mach port object is also not related to the Carbon graphics port primitive (`GrafPort`).

Library Managers and Executable Formats

A runtime environment (or, simply, runtime) is a set of conventions that determines how code and data are loaded into memory and managed. Mac OS X supports two primary runtime environments—dyld (dynamic link editor) and CFM (Code Fragment Manager). One of the thorny issues raised by multiple runtimes on one system is how to allow, for example, code prepared for one runtime to access code prepared for another. This section discusses the issue and describes the technology Apple has developed for bridging between them. It also explains Apple's position on the runtime approaches it recommends to developers.

This section provides a comparative overview of the dyld and CFM runtimes as well as the executable formats of the code and data they operate upon. For a more detailed discussion of the CFM-based runtime environment, see the Carbon documentation on the Code Fragment Manager, especially the chapter "CFM-Based Runtime Architecture." In this book, see "Dynamic Shared Libraries" (page 144) in the chapter ""Frameworks" (page 139)" for a description of the dynamic link editor. Also, see "CFM Executables" (page 229) for a description of how to prepare a CFM executable for Mac OS X.

Comparing the Runtime Environments

A CFM-based application cannot *directly* call a function in a dyld-based framework, and the reverse is also true. In order to understand this restriction—and Apple's solution—you must first understand the major differences between the two environments.

CFM and dyld

The Code Fragment Manager (CFM) and the dynamic link editor (dyld) are library managers. (Other terms might also be applicable but, for the sake of this discussion, “library manager” suffices). A library manager is responsible for mapping one or more containers (or modules) of code and data into memory and preparing them for execution. It prepares them for execution primarily by attempting to resolve references to symbols defined externally. These symbols are typically defined in shared libraries that the container links with at build time.

The major difference in the behavior of the dyld and CFM library managers is *when* they resolve these references and bind them to addresses in the appropriate libraries. CFM takes a static approach; it prepares each container of code and data (called a fragment) as a unit (called a closure). At build time, CFM finalizes the executable by determining where the various referenced symbols will exist at runtime. The dyld library manager, on the other hand, attempts to resolve all undefined symbols at runtime. More specifically, symbols are resolved only as they are referenced during program execution. The dyld manager links code modules in a dynamic shared library only as they are needed.

PEF and Mach-O

Both the dyld and CFM library managers expect the container of code and data that they prepare for execution to be in a certain executable file format. The executable format is a packaging convention for machine-ready (executable) code. For CFM, this format is called PEF (Preferred Executable Format) and for dyld, the format is called Mach-O (Mach object-file format).

PEF and Mach-O are similar in many respects. They both define sections (or segments) for code, global data, non-constant data, and so on. Where they primarily differ is their allowance for multiple containers. PEF is a format for a container (a fragment) that maps one-to-one to an executable. In the dyld world, however, an executable can be composed from multiple Mach-O containers (object files).

Code-Generation Models

Although they are significant, the major differences discussed so far between library managers and their executable formats do not explain why a CFM-based program cannot directly call a function in a dyld-based library. The real source of incompatibility between the CFM runtime and the dyld runtime is the different external calling conventions used by their code-generation models. The differences affect the representation of C function pointers and the way global data is accessed.

- The CFM code-generation model uses a pointer to a TVector as the basis for function pointers. It accesses global data indirectly via the R2 register, using it as a base pointer to the global data. This method of access is known as TOC.
- The dyld code-generation model uses a simple pointer to code as the basis for function pointers. It accesses global data relative to code, using an offset from a base address. This method of access is known as GOT.

Vector Libraries

All system frameworks in Mac OS X are based on dyld and Mach-O. Some of these frameworks contain Carbon APIs. Therefore, if you have a CFM-based Carbon application or library, your code needs to call functions in these system frameworks. Apple has made it possible for CFM-based code to call functions in a dyld-based framework through a technology called a vector library. A vector library functions as a bridge for a system framework that contains Carbon APIs. Part of this bridge is a vector or jump table that provides the "glue" code to handle the differences in code-generation models. A CFM-based client (application or library) can use these vector libraries and thereby access the Carbon APIs in the associated dyld-based framework.

Carbon developers don't have to do anything special to access code in system frameworks, as long as that code is defined as part of Carbon. To take advantage of the bridging technology of the vector libraries, they need only link against the stub libraries found in the CarbonLib SDK.

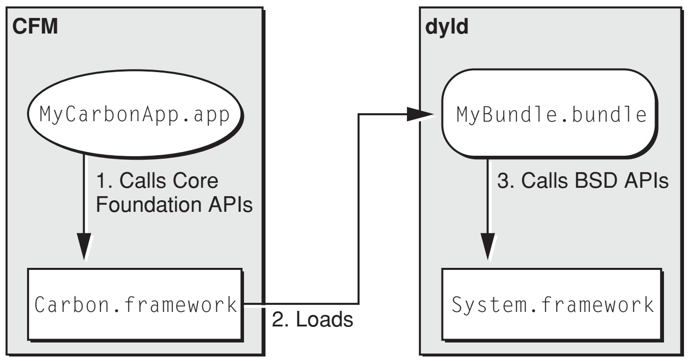
Note that vector libraries do not bridge in the other direction—from a dyld application or framework into a CFM library. It is possible to call from dyld to CFM using a CFBundle, but this solution is not appropriate for all situations. In general, if you want a library to be available to all of the Mac OS X execution environments, you should build it as a dyld-based library.

CFM Executable and Non-Carbon APIs

There may be occasions when a CFM-based application or library wants to call into a system framework that does not contain any Carbon APIs. A good example of such a framework is `System.framework`, which implements many of the POSIX kernel-environment APIs. Mac OS X supplies no vector libraries for this purpose.

The system does provide another mechanism for accessing non-Carbon APIs in system frameworks: the bundle. You can create a dyld-based bundle that links with a non-Carbon framework and is therefore able to directly call the framework's functions. A CFM application can then use the Carbon bundle APIs (specifically Core Foundation Bundle Services) to load the bundle and use it to call into the non-Carbon framework. Figure 14-2 illustrates how this is done.

Figure 14-2 A Carbon application calling BSD system routines



Should You Use CFM or dyld?

Experienced developers understand that if you want to maximize an application's performance for a particular platform, you must optimize your code for that platform. Mac OS X is natively a dyld platform. After all, the system frameworks—even the ones with Carbon APIs—are based on dyld and Mach-O. In fact, the Code Fragment Manager itself is built on top of dyld technology. For this reason, Apple strongly encourages developers to use dyld and Mach-O for their programs.

For compatibility reasons, CFM-based programs are supported on Mac OS X. However, Apple encourages the use of the application packaging scheme described in the chapter "Application Packaging" (page 131)) to build application bundles containing multiple executable formats. If you are developing a Carbon application, and want to maximize performance on both Mac OS X and Mac OS 9, then you should compile the same set of source code for both runtime environments—one for dyld, the other for CFM—then optimize each executable for its intended platform. In this way, your application can take advantage of the native runtime environment on both platforms.

APPENDIX A

Information Property List Keys

This appendix describes the keys you can specify in property list files for bundles and packages.

Bundle Keys

Information about a bundle is specified using property lists inside the bundle directory. The Finder and system APIs use this information when several situations. Bundles support the following types of keys:

- Core Foundation keys—specify general bundle properties
- Cocoa-specific keys—specify properties for bundles written using Cocoa
- Finder-specific keys—specifies information used by the Finder and file system
- Launch Services keys—specifies information used by Launch Services.

Standard Bundle Keys

Mac OS X provides a set of core keys for specifying information about a bundle. Some of these keys are given default values by the integrated development environment. Table A-1 lists the keys to specify for bundles.

Table A-1 Summary of standard keys

Key	Type	Required	Summary
CFBundleDevelopmentRegion	`String`	No	The native region for the bundle. Usually corresponds to the native language of the author.
CFBundleDisplayName	`String`	No	The localized display name of the bundle.
CFBundleDocumentTypes	`Array`	No	An array of dictionaries describing the document types supported by the bundle
CFBundleExecutable	`String`	Yes	Name of the bundle executable file.
CFBundleGetInfoHTML	`String`	No	A string for displaying richer content in the Finder's Get Info panel
CFBundleGetInfoString	`String`	No	A string for display in the Finder's Get Info panel
CFBundleHelpBookFolder	`String`	No	The name of the folder containing the bundle's help files.
CFBundleHelpBookName	`String`	No	The name of the help file to display when help is launched for the bundle.
CFBundleIconFile	`String`	Yes	File name for icon image file
CFBundleIdentifier	`String`	Yes	Unique identifier string for the bundle. This string should be in the form of a java package name, for example `com.apple.myapp`
CFBundleInfoDictionaryVersion	`String`	Yes	Version information for the `Info.plist` format.

Table A-1 Summary of standard keys (continued)

Key	Type	Required	Summary
CFBundleName	`String`	Yes	The short display name of the bundle
CFBundlePackageType	`String`	Yes	The four-letter code identifying the bundle type
CFBundleShortVersionString	`String`	Yes	The marketing-style version string for the bundle
CFBundleSignature	`String`	Yes	The four-letter code identifying the bundle creator
CFBundleURLTypes	`Array`	No	An array of dictionaries describing the URL schemes supported by the bundle
CFBundleVersion	`String`	Yes	Build number of the executable.

CFBundleDevelopmentRegion

The CFBundleDevelopmentRegion key specifies a string value identifying the native region for the bundle. Usually, this value corresponds to the native language of the person who wrote the bundle. This value is used as the last resort if a resource cannot be located for the user's preferred region or language.

CFBundleDisplayName

The CFBundleDisplayName key specifies a string value identifying the display name of the bundle. This is the name that the Finder and other user interface elements display to the user. It does not necessarily correspond to the name of the bundle in the file system. This key can be localized by including it in the `InfoPlist.strings` file of the appropriate `.lproj` subdirectory. If you localize this key, you should also provide a localized version of the CFBundleName key.

See "Localizing File System Names" (page 176) for more information on display names.

CFBundleDocumentTypes

The CFBundleDocumentTypes key contains an array of dictionaries, each of which contains information about a document type supported by the application. Each dictionary is called a type-definition dictionary and contains keys used to define the document type. Table A-2 lists the keys that are supported in a type-definition dictionary:

Table A-2 Keys for CFBundleDocumentTypes dictionaries

Key	Type	Description
CFBundleTypeExtensions	`Array`	This key contains an array of filename extensions that map to this type. To open documents with any extension, specify an extension with a single asterisk "*". This key is required.
CFBundleTypeIconFile	`String`	This key specifies the name of the icon file to be used when displaying documents of this type. The icon filename can have an extension or be without one. If it is without an extension, the system appends an extension appropriate to the platform (for example, `.icns` in Mac OS 9).
CFBundleTypeName	`String`	This key contains the abstract name for the document type and is used to refer to the type. This key is required and can be localized by including it in the corresponding `InfoPlist.strings` files. This value is the main way to refer to a type and it is recommended that you use a Java-style package identifier to ensure its uniqueness. If the type is a common Clipboard type supported by the system, you can use one of the standard types listed in the NSPasteboard class description.

Table A-2 Keys for CFBundleDocumentTypes dictionaries (continued)

Key	Type	Description
CFBundleTypeOSTypes	`Array`	This key contains an array of four-letter type codes that map to this type. To open documents of any type, specify the four - letter type code '`****`'. This key is required
CFBundleTypeRole	`String`	This key specifies the application's role with respect to the type. The value can be `Editor`, `Viewer`, `Printer`, `Shell`, or `None`. See "Document Configuration" (page 199) for descriptions of these values. This key is required.
NSDocumentClass	`String`	This key specifies the NSDocument subclass used to instantiate instances of this document. Used for Cocoa applications only.
NSExportableAs	`Array`	This key specifies an array of other types that documents of this type can be exported as (write-only types). Used for Cocoa applications only.

CFBundleExecutable

The CFBundleExecutable key identifies the name of the bundle's main executable file. For an application, this is the application executable. For a loadable bundle, it is the binary that will be loaded dynamically by the bundle. For a framework, it is the shared library for the framework. Project Builder automatically adds this key to the `Info.plist` file of appropriate projects.

For frameworks, the executable name is required to be the same as the framework name for launch-performance reasons. The executable name should not include any extension that may be used on various platforms.

Important
You must include a valid CFBundleExecutable key in your bundle's `Info.plist` file. Mac OS X uses this key to locate the bundle's executable or shared library in cases where the user renames the application or bundle directory.

CFBundleGetInfoHTML

The CFBundleGetInfoHTML key identifies the human-readable HTML string displayed in the Info window of the bundle. You can specify this key-value pair instead of the plain text CFBundleGetInfoString if you want a richer representation in the Info window. You can localize this string by including it in the `InfoPlist.strings` file of the appropriate `.lproj` directory.

If the CFBundleGetInfoString and CFBundleGetInfoHTML keys are both present, the CFBundleGetInfoHTML key is used.

CFBundleGetInfoString

The CFBundleGetInfoString key identifies a human-readable plain text string displayed in the Info window of the bundle. (This string was also known as the long version string in Mac OS 9). The format of the key should conform to the long version string of Mac OS 9, for example, "2.2.1, © Great Software, Inc, 1999". You can localize this string by including it in the `InfoPlist.strings` file of the appropriate `.lproj` directory.

If the CFBundleGetInfoHTML is also present, this key is not used.

CFBundleHelpBookFolder

The CFBundleHelpBookFolder key identifies the folder containing the bundle's help files. Help is usually localized to a specific language, so the folder specified by this key represents the folder name inside the `.lproj` directory for the selected language.

CFBundleHelpBookName

The CFBundleHelpBookName key identifies the main help page for your application. This key identifies the name of the Help page, which may not correspond to the name of the HTML file. The Help page name is specified in the `CONTENT` attribute of the help file's `META` tag.

CFBundleIconFile

The CFBundleIconFile key identifies the file containing the icon for the bundle. The filename you specify does not need to include the `.icns` extension, although you can. The Finder looks for the icon file in the `Resources` directory of the bundle.

If your bundle uses a custom icon, you must specify this property. If you do not specify this property, the Finder (and other applications) display your bundle with a default icon.

CFBundleIdentifier

The CFBundleIdentifier key specifies a unique identifier string for the bundle. This identifier should be in the form of a Java-style package name, for example `com.apple.myapp`. The bundle identifier can be used to locate the bundle at runtime. The preferences system uses this string to identify applications uniquely.

CFBundleInfoDictionaryVersion

The CFBundleInfoDictionaryVersion key identifies the current version of the property list structure. This key exists to support future versioning of the `Info.plist` format. Project Builder generates this key automatically when you build a bundle.

CFBundleName

The CFBundleName key identifies the short name of the bundle. This name should be less than 16 characters long and be suitable for displaying in the menu and the About box. This key can be localized by including it in the `InfoPlist.strings` file of the appropriate `.lproj` subdirectory. If you localize this key, you should also provide a localized version of the CFBundleDisplayName key.

CFBundlePackageType

The CFBundlePackageType key identifies the type of the bundle and is analogous to the Mac OS 9 file type code. The value for this key consists of a four-letter code. For applications, this code is `'APPL'`; for frameworks, it is `'FMWK'`; for loadable bundles, it is `'BNDL'`. For loadable bundles, you can also choose a type code that is more specific than `'BNDL'` if you want.

CFBundleShortVersionString

The CFBundleShortVersionString key identifies the marketing-style version of the bundle. The marketing-style version string usually displays the major and minor version of the bundle. This string is usually of the form *n.n.n* where *n* represents a number. The first number is the major version number of the bundle. The second and third numbers are minor revision numbers. The value of this key is displayed in the default About box for Cocoa applications.

The value for this key differs from the value for the CFBundleVersion key, which identifies a specific build number. The CFBundleShortVersionString value represents a more formal version that does not change with every build.

CFBundleSignature

The CFBundleSignature key identifies the creator of the bundle and is analogous to the Mac OS 9 file creator code. The value for this key consists of a four-letter code that is specific to each bundle.

CFBundleURLTypes

The CFBundleURLTypes key contains an array of dictionaries describing the URL schemes supported by the application. The purpose of this key is similar to that of the CFBundleDocumentTypes key, but it describes URL schemes instead of document types. Each dictionary entry corresponds to a single URL scheme. Table A-3 lists the keys to use in each dictionary entry.

Table A-3 Keys for CFBundleURLTypes dictionaries

Key	Type	Description
CFBundleTypeRole	`String`	This key specifies the application's role with respect to the URL type. The value can be `Editor`, `Viewer`, `Printer`, `Shell`, or `None`. See "Document Configuration" (page 199) for descriptions of these values. This key is required.
CFBundleURLIconFile	`String`	This key contains a string entry with the name of the icon image file (minus the extension) to be used for this URL type.
CFBundleURLName	`String`	This key contains a string entry with the abstract name for this URL type. This is the main way to refer to a particular type. To ensure uniqueness, it is recommended that you use a Java-package style identifier. This name is also used as a key in the `InfoPlist.strings` file to provide the human-readable version of the type name.
CFBundleURLSchemes	`Array`	This key contains an array of the URL schemes handled by this type. Examples of URL schemes include `http`, `ftp`, and so on.

CFBundleVersion

The CFBundleVersion key specifies an application-specific string for identifying the build number. The value of this key typically changes between builds and is displayed in the Cocoa About panel in parenthesis.

To specify the version information of a released bundle, use the CFBundleShortVersionString key. See "CFBundleShortVersionString" (page 279) for more information.

Application-Specific Keys

Table A-4 lists the keys that are applicable to Application bundles only:

Table A-4 Application-specific keys

Key	Type	Required	Summary
CFAppleHelpAnchor	`String`	No	The bundle's initial HTML help file.
NSAppleScriptEnabled	`String`	No	Specifies whether AppleScript is enabled.
NSHumanReadableCopyright	`String`	Yes	A copyright string used for display in dialog boxes.
NSJavaNeeded	`Boolean` or `String`	No	Specifies whether the program requires a running Java VM.
NSJavaPath	`Array`	No	An array of paths to classes whose components are preceded by NSJavaRoot.
NSJavaRoot	`String`	No	The root directory containing the java classes.
NSMainNibFile	`String`	Yes	The name of an application's main nib file.
NSPrincipalClass	`String`	Yes	The name of the bundle's main class.
NSServices	`Array`	No	An array of dictionaries specifying the services provided by an application.

CFAppleHelpAnchor

The CFAppleHelpAnchor key identifies the name of the bundle's initial HTML help file, minus the `.html` or `.htm` extension. This file is located in the bundle's localized resource directories or, if non-localized, directly under the `Resources` directory.

NSAppleScriptEnabled

The NSAppleScriptEnabled key identifies whether the application is scriptable. Set the value of this string to "Yes" if your application supports AppleScript.

NSHumanReadableCopyright

The NSHumanReadableCopyright key contains a string with copyright information for the bundle. You can load this string and display it in an About dialog box. This key is usually in the `InfoPlist.strings` file because it needs to be localized.

NSJavaNeeded

The NSJavaNeeded key contains a Boolean value that determines whether the Java VM must be loaded and started up prior to executing the bundle code. You can also specify a string type with the value "YES" instead of a Boolean value if desired.

NSJavaPath

The NSJavaPath key contains an array of paths. Each path points to a Java class. The path can be either an absolute path or a relative path from the location specified by the NSJavaRoot key. The development environment (or, specifically, its jamfiles) automatically maintains the values in the array.

NSJavaRoot

The NSJavaRoot key contains a string identifying a directory. This directory represents the root directory of the application's Java class files.

NSMainNibFile

The NSMainNibFile key contains a string with the name of the application's main nib file (minus the `.nib` extension). A nib file is an Interface Builder archive containing the description of a user interface along with any connections between the objects of that interface. The main nib file is automatically loaded when an application is launched. Mac OS X looks for a file whose name matches the name of the application.

NSPrincipalClass

The NSPrincipalClass key contains a string with the name of a bundle's principal class. The principal class should have a central relation to other classes in the bundle. For applications, this name is the application name by default.

NSServices

The NSServices key contains an array of dictionaries specifying the services provided by an application. Table A-5 lists the keys for specifying a service:

Table A-5 Keys for NSServices dictionaries

Key	Type	Description
NSPortName	`String`	This key specifies the name of the port your application monitors for incoming service requests.
NSMessage	`String`	This key specifies the name of the instance method to invoke for the service. In Objective-C, the instance method must be of the form `messageName:userData:error:`. In Java, the instance method must be of the form `messageName(NSPasteBoard,String)`.
NSSendTypes	`Array`	This key specifies an array of data type names that can be read by the service. The NSPasteboard class description lists several common data types. You must include this key, the NSReturnTypes key, or both.

Table A-5 Keys for NSServices dictionaries (continued)

Key	Type	Description
NSReturnTypes	`Array`	This key specifies an array of data type names that can be returned by the service. The NSPasteboard class description lists several common data types. You must include this key, the NSSendTypes key, or both.
NSMenuItem	`Dictionary`	This key contains a dictionary that specifies the text to add to the Services menu. The only key in the dictionary is called `default` and its value is the menu item text. This value must be unique. You can use a slash character "/" to specify a submenu. For example, `Mail/Send` would appear in the Services menu as a menu named Mail with an item named Send.
NSKeyEquivalent	`Dictionary`	This key is optional and contains a dictionary with the keyboard equivalent used to invoke the service menu command. Similar to NSMenuItem, the only key in the dictionary is called `default` and its value is a single character. Users invoke this keyboard equivalent by pressing the Command and Shift key modifiers along with the character.
NSUserData	`String`	This key is an optional string that contains a value of your choice.
NSTimeout	`String`	This key is an optional numerical string that indicates the number of milliseconds Services should wait for a response from the application providing a service when a response is required.

Launch Services Keys

The Launch Services keys restrict how applications are launched in Mac OS X. These keys apply both to CFM and Mach-O executables. For more information on CFM versus Mach-O executables, see the section "CFM Executables" (page 229) in the chapter "Installation and Integration" (page 223). Table A-6 lists the keys recognized by Launch Services.

Table A-6 Launch Services keys

Key	Type	Required	Summary
LSBackgroundOnly	`String`	No	Specifies whether the application runs only in the background. (Mach-O applications only)
LSPrefersCarbon	`String`	No	Specifies whether an application prefers running in the Carbon environment.
LSPrefersClassic	`String`	No	Specifies whether an application prefers running in the Classic environment.
LSRequiresCarbon	`String`	No	Specifies whether the application must run as a Carbon application.
LSRequiresClassic	`String`	No	Specifies whether the application must run in the Classic environment.
LSUIElement	`String`	No	Specifies whether the application is a user-interface element, that is, an application that should not appear n the Dock or Force Quit window.

LSBackgroundOnly

If this key exists and is set to "1", Launch Services runs the application in the background only. You can use this key to create faceless background applications. You should also use this key if your application uses higher-level frameworks that connect to the window server, but are not intended to be visible to users. Background applications must be compiled as Mach-O executables. This option is not available for CFM applications.

You can also specify the type of this key as `Boolean` or `Number`. However, these type values are only supported in Mac OS X 10.2 or later.

LSPrefersCarbon

If this key is set to “1,” the Finder displays the “Open in the Classic environment” control in the application’s Get Info panel, and leaves the control unchecked by default. The user can modify this control to launch the application in the Classic environment if desired.

You can also specify the type of this key as `Boolean` or `Number`. However, these type values are only supported in Mac OS X 10.2 or later. If you include this key in your property list, do not include the LSPrefersClassic, LSRequiresCarbon, or LSRequiresClassic keys.

LSPrefersClassic

If this key is set to “1,” the Finder displays the “Open in the Classic environment” control in the application’s Get Info panel, and puts a check mark in the control by default. The user can modify this control to launch the application in the Carbon environment if desired.

You can also specify the type of this key as `Boolean` or `Number`. However, these type values are only supported in Mac OS X 10.2 or later. If you include this key in your property list, do not include the LSPrefersCarbon, LSRequiresCarbon, or LSRequiresClassic keys.

LSRequiresCarbon

If this key is set to “1,” Launch Services runs the application in the Carbon environment only. Use this key if your application should not be run in the Classic environment.

You can also specify the type of this key as `Boolean` or `Number`. However, these type values are only supported in Mac OS X 10.2 or later. If you include this key in your property list, do not include the LSPrefersCarbon, LSPrefersClassic, or LSRequiresClassic keys.

LSRequiresClassic

If this key is set to "1," Launch Services runs the application in the Classic environment only. Use this key if your application should not be run in the Carbon compatibility environment.

You can also specify the type of this key as `Boolean` or `Number`. However, these type values are only supported in Mac OS X 10.2 or later. If you include this key in your property list, do not include the LSPrefersCarbon, LSPrefersClassic, or LSRequiresCarbon keys.

LSUIElement

If this key is set to "1", Launch Services runs the application as a user interface element. User interface elements do not appear in the Dock or in the Force Quit window. Although they typically run as background applications, they can come to the foreground to present a user interface if desired. A click on a window belonging to a user interface element brings that application forward to handle events.

The Dock and loginwindow are two applications that run as user interface elements.

Application Package Keys

The purpose of application packaging is to make an application a self-contained entity and hide its content from the user. However, applications often have files that the user may want to manipulate. For example, the user may want to add or remove a plug-ins, localized resources, and so on. Developers can specify the items that are allowed to be manipulated by the user in the `Info.plist` file. The Finder then makes these items available from the bundle's Info panel, allowing the user to browse these items and remove or add items as desired.

CFBundleInstallerInfo

The root key for application package information is the **CFBundleInstallerInfo** key. This key specifies a dictionary that contains the keys shown in Table A-7. The Required column lists the keys that you must have to support this feature.

Table A-7 Application packaging keys

Key	Type	Required	Summary
APInstallerURL	`String`	Yes	A URL-based path to the files you want to install.
APFiles	`Array`	Yes	An array of dictionaries describing the files or directories that can be installed.

APInstallerURL

The APInstallerURL key identifies the base path to the files you want to install. You must specify this path using the form `file://localhost/path/`. All installed files must reside within this directory.

APFiles

The APFiles key specifies a dictionary with or more files you want to install. Each dictionary entry can contain the description for a file or directory. You can nest the APFiles key inside itself to specify files inside of a directory. Table A-8 lists the keys for specifying information about a single file or directory.

Table A-8 Keys for APFiles dictionary

Key	Type	Description
APFileDescriptionKey	`String`	A short description of the item to display in the Finder's Info window
APDisplayedAsContainer	`String`	If "Yes" the item is shown with a folder icon in the Info panel; otherwise, it is shown with a document icon
APFileDestinationPath	`String`	Where to install the component as a path relative to the application bundle
APFileName	`String`	The name of the file or directory
APFileSourcePath	`String`	The path to the component in the application package relative to the APInstallerURL path.
APInstallAction	`String`	The action to take with the component: "Copy" or "Open"

APPENDIX B

Document Revision History

Table B-1 describes the revisions to Inside Mac OS X: System Overview.

Table B-1 Document Revision History

Date	Notes
Feb 2003	Corrected several documentation bugs.
July 2002	Updated the book with new content covering the features of Mac OS X 10.2.
	Revamped the Booting and Logging In chapter.
	Updated the File System chapter to include new information about directories and localization.
	Created a new appendix to hold the property-list keys targeted for info.plist files.
	Fixed numerous documentation bugs.

A P P E N D I X   B

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**.

active window The front-most 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 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 referenced 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."

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.

BSD Berkeley Software Distribution. Formerly known as the Berkeley version of UNIX, BSD is now simply called the BSD operating system. The BSD portion of Mac OS X is based on 4.4BSD Lite 2 and FreeBSD, a "flavor" of 4.4BSD.

buffered window A window with a memory buffer into which all drawing is rendered. All graphics are first drawn in the buffer, 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 programming interfaces derived from earlier versions of the Mac OS. The Carbon APIs have 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, Mac OS 9, and all versions of Mac OS 8 later than Mac OS 8.1.

CFM Code Fragment Manager, the library manager and code loader for processes based on PEF (Preferred Executable Format) object files (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 Power PC and 68K 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. The 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's 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 with programming interfaces in both Java and Objective-C. 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 a scanner, a video display, and a printer) 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 8 and 9 are cooperative multitasking environments. See also **preemptive multitasking**.

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 Open Source technology.

demand paging An operating system facility that causes pages of data to be brought 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 applicability 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 folder that the Finder presents to users as if it were a file. In other words, the Finder hides the contents of the folder from users. This opacity discourages users from inadvertently (or intentionally) altering the contents of the bundle.

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, for storing data in files, and for specifying locations of files. See also **volume format**.

firewall Software (or a computer running such software) that prevents unauthorized access to a network by users outside of 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 a 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 folders themselves. HFS is a two-fork volume format.

HFS+ Hierarchical File System Plus. The Mac OS Extended file-system format. This file-system 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.

host The computer that's running (is host to) a particular program. The term is usually used to refer to a computer on a network.

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 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.

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 datum in a data structure such as a dictionary.

localization The adaptation of a software product, including 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. 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 the program to build 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 8 and 9 do 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. Mac OS 8 and 9 use cooperative multitasking.

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

nonretained window A window without an off-screen 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 are a way a process can 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 APIs. Some Cocoa objects wrap Carbon APIs 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.

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 the transfer of 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 is reserved for a special character that says if 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' 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).

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 to illuminate each of these coordinates with. 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.

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 where 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.

SMP Symmetric multiprocessing. A feature of an operating system in which two or more processors are managed by one kernel, sharing the same memory, 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 have spoolers.

subframework A public framework that packages a specific Apple technology, such as Apple events or Open Transport. 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. Unlike ASCII, which defines 128 distinct characters typically represented in 8 bits, there are as many as 65,536 distinct Unicode 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 they makes it possible to change any element of the picture at any time since 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 in 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 same facilities usually provided by RAM. The virtual-memory manger in Mac OS X provides 32-bit (minimum) protected address space for each task and facilitates efficient sharing of that address space.

volume A storage device or a portion of that 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 lengths of filenames. See also **file system**.

window server A system-wide 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.

Index