INSIDE MAC OS X

System Overview

Preliminary

May 2000

 Apple Computer, Inc.
© 2000 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Computer, Inc.
Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Chapter 1 About This Book 9

Why Read This Book 9
Further Investigations 11
  Installed Developer Documentation 11
  Other Apple Publications 12
  Information on the Web 12

Chapter 2 System Technologies 15

The User Experience 16
  Aqua 17
  The Desktop and the Finder 18
  Application Support 19
  Multiple Users 20
  Internationalization 21
  Application Extensibility 22
  Exported Application Services 23
  Other Parts of the User Experience 24
Darwin 24
  Mach 24
  BSD 26
  Device-Driver Support 26
  Networking Extensions 27
  File Systems 27
  Darwin and Open Source Development 29
Graphics and Imaging 29
  Quartz 29
  QuickDraw 31
  OpenGL 31

Preliminary © Apple Computer, Inc. May 2000

QuickTime 32
Printing 32
Apple Type Solution 33
Networking and the Internet 34
Media Types 34
Standard Protocols 35
Legacy Network Services and Protocols 36
Routing and Multihoming 36
Personal File and Web Services 37
Advanced Hardware Features 37
USB 37
FireWire 38
Velocity Engine 38

Chapter 3 System Architecture 39

A Layered Perspective 40
Application Environments 45
Carbon 45
Cocoa 47
Java 49
The Graphics and Windowing Environment 51
Core Graphics Services 53
Core Graphics Rendering 54
The Printing System 56
User Interface 56
Architecture Summary 57
Printer Discovery 59
The Printing Process 60
Other Application Services 61
Process Manager 61
Carbon Event Manager 61
The Pasteboard 62

Preliminary © Apple Computer, Inc. May 2000

Core Services 62
Carbon Managers 62
Core Foundation 64
Apple Events 66
Open Transport 67
Tracking a User Event 67

Chapter 4 Bundles 71

Benefits of Using Bundle 72
Anatomy of a Bundle 73
The Finder and Bundles 77
Types of Bundles 78
An Application's Main Bundle 79
Framework Bundles 80
Loadable Bundles and Dynamic Linking 80
Localized Resources 81
Localized Character Strings 82
Search Algorithm 82
Bundles and the Resource Manager 84

Chapter 5 Application Packaging 87

An Application Is a Bundle 87
Application Frameworks, Libraries, and Helpers 89
Private Frameworks 90
Shared Frameworks and the Central Directory 91
Other Shared Application Code 92
Applications and Loadable Bundles 93
User Resources in Applications 94
Application Help 94
Application Preferences 95
Document Resources 96

Preliminary © Apple Computer, Inc. May 2000

Chapter 6 Frameworks 97

The Framework as a Library Package 98
The Internal Structure of Frameworks 99
Standard Locations for Frameworks 101
Dynamic Shared Libraries 102
Framework Versioning 104
Major Versions 105
Minor Versions 106
Versioning Summary and Guidelines 107
Guidelines for Major Versioning 108

Chapter 7 Umbrella Frameworks 111

Kinds of Frameworks 112
The Purpose of Umbrella Frameworks 113
Linking and Including Guideline 115
The Structure of an Umbrella Framework 116
Restrictions on Subframework Linking 118

Chapter 8 The Desktop and the File System 119

The Role of the Desktop 120
Desktop Interfaces to Applications 121
Information Property Lists 121
Information Stored by the Desktop 122
Collecting Application Information 122
The Desktop Folder 123
Finder Attributes 123
The Handling of Applications and Documents 124
The Finder and File Operations 126
Copy and Move Operations 126
Management of Aliases and Symbolic Links 127
File-System Topics 127
Aliases and Symbolic Links 127
Resource Forks 128

Preliminary © Apple Computer, Inc. May 2000

Chapter 9 Software Configuration 131

Property Lists 131
Information Property Lists 132
  Document Configuration 133
  An Example 134
  Standard Keys 137
  Finder Keys 139
The Preferences System 141
  How Preferences Are Stored 142
  Preference Domains 143
  The defaults Utility 144

Chapter 10 Inter-Environment Issues 145

Tasks and Processes 145
Threading Packages 146
  Layering Details 148
  Usage Guidelines 148
Interprocess Communication 149
Library Managers and Executable Formats 151
  Comparing the Runtime Environments 151
    CFM and dyld 151
    PEF and Mach-O 152
    Code-Generation Models 152
  Vector Libraries 153
  CFM Executable and Non-Carbon APIs 153
  Should You Use CFM or dyld? 154

Glossary 155

Index 163

Preliminary © Apple Computer, Inc. May 2000

CHAPTER 1

About This Book

Preliminary documentation: This version of *Inside Mac OS X: System Overview* is in a preliminary stage of completion. The final version will contain new and updated material. This preliminary version of the book is offered as background information for developers installing the DP4 (Developer Preview 4) version of Mac OS X.

With Mac OS X, Apple is reasserting its leadership not only in operating systems but in the advanced technologies and design sensibility that are the hallmarks of that operating system. 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 phenomenal 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 and value 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 or UNIX or Windows or any other platform—especially Open Source developers working with Darwin—you are apt to find something of value in this book.

Preliminary © Apple Computer, Inc. May 2000

About 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.
- **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 Desktop and the File System**. Discusses the interfaces between the Desktop and applications. It explains how the Finder (the major component of the Desktop) handles various tasks, such as determining application ownership of documents and copying files between volumes of different formats. Finally, this section discusses topics related to the file system, such as the standard directory layout, resource forks, and aliases versus symbolic links.
- **Software Configuration**. Describes the basic mechanisms for configuring applications and other bundles and for handling user preferences.
- **Inter-Environment Issues**. Discusses some of the programming issues arising from multiple application environments and layered architecture in Mac OS X.

Future versions of the book will include additional material.

Preliminary © Apple Computer, Inc. May 2000

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 on 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 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 on applications such as Project Builder and Interface Builder is installed with the application. When users request it from the Help menu, the application launches Help Viewer to display it.
- Example code. A variety of sample programs are installed in `/System/Developer/Examples` showing you how to perform common tasks using the primary Mac OS X application environments—Carbon, Cocoa, and Java
- `/System/Developer/Documentation`. All information that is not specific to frameworks or development applications is installed here. The installer also creates in this location symbolic links to the framework documentation.

Preliminary © Apple Computer, Inc. May 2000

Apple's developer documentation uses Apple Help, and specifically the Help Viewer, as a presentation and access mechanism. To view and search developer documentation you use a special user interface for the Help Viewer that bears the title "Developer Center." To access the Developer Center:

1. Choose Help Center from the Desktop's Help menu.

   The Help Center is used for accessing user documentation.

2. Click the Developer Center link on the first (home) page of the Help Center.
3. To return to the Help Center, click the Help Center link on the home page of the Developer Center.

The home page of the Developer Center lists links to the "books" that are currently installed. The behavior of the Help Viewer is very much like a typical browser. However, links to external URLs will open those URL resources in your preferred Web browser.

The scope of a search using the Developer Center is determined by your current location within the set of books. If you are browsing through a particular book—say, Core Foundation Reference—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 Center. If you perform a search from the home page of the Developer Center, the Help Viewer searches the indexes of all books belonging to the Developer Center.

Other Apple Publications

Apple is planning a series of *Inside Mac OS X* books. This book, the *System Overview*, is the first of that series. You can obtain other books in this series (as they become available) using the publish-on-demand arrangement Apple has with Fatbrain.com.

To obtain your printed copy of an *Inside Mac OS X* book, use your Web browser to access the page at `www.fatbrain.com/documentation/apple`. Then follow the directions. The book should be delivered to you within a few business days.

Information on the Web

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

Preliminary © Apple Computer, Inc. May 2000

- Apple Product Information (`www.apple.com/macosx`). Provides general information on Mac OS X.
- Apple Developer Connection—Developer Documentation (`developer.apple.com/techpubs`). Features the same documentation that is installed on Mac OS X, except that often the documentation is more up-to-date. Also includes legacy documentation.
- AppleCare Tech Info Library (`til.info.apple.com`). Contains technical articles, tutorials, FAQs, technical notes, and other information.
- Apple Developer Connection—Mac OS X (`developer.apple.com/macosx`). Offers SDKs, release notes, product notes and news, and other resources and information related to Mac OS X.

Preliminary © Apple Computer, Inc. May 2000

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 new and some 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 sports a sparkling new 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.

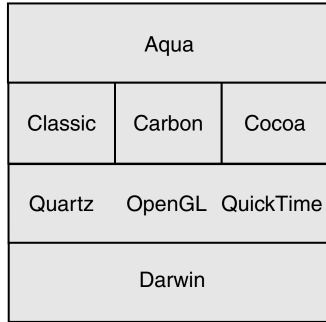
From a functional perspective, the most important components of Mac OS X are

- Aqua, the human-interface design behind the user's experience
- the application environments Classic, Carbon, and Cocoa
- the windowing and graphics system, as implemented by Quartz, QuickTime, and OpenGL
- Darwin, the advanced core of the operating system

Preliminary © Apple Computer, Inc. May 2000

Figure 2-1 depicts the general dependencies between these components. The rest of this section 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 what Macintosh users have become comfortably familiar with. But it is also different in important and even spectacular ways. It features a radical new design for the user interface, a new infrastructure for localizing the interface, a new way to add application features dynamically, and both new and familiar mechanisms for exporting and accessing the services of other applications.

And, of course, the new user experience draws from the benefits obtained through the core of the operating system (see "Darwin" (page 24)). A Macintosh remains stable even when an application crashes, and no single application or task can now hog 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.

Preliminary © Apple Computer, Inc. May 2000

Aqua

When Apple designed Aqua, the new 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 Macintosh you've ever seen. As "aqua" suggests, the properties of water infuse the lucid feel 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 seem to have depth with their drop shadows, minimized windows swoop into their dock icon like a genie into her bottle.

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 box to print or save a document, you had to know which document the dialog box was for, even though you might have many documents open at a time. Mac OS X introduces a new type of dialog box that attaches to a document and makes their relationship clear. These new dialog boxes appear to slide out from underneath the window title, and their translucent quality makes them look as though they're floating above the document. Also, these dialog boxes are no longer modal, hijacking your computer and demanding your immediate attention. You can proceed to other tasks before dismissing the dialog box—without having to interrupt what you're doing.

Single Window Mode is a optional feature users can turn on to help them manage screen space. It applies the concept of a single window to the whole operating system, eliminating the need to search for or arrange windows. When you're in Single Window Mode the computer makes the current window the active window

Preliminary © Apple Computer, Inc. May 2000

and automatically hides all the other open windows. When you want to work on another document or application, Mac OS X automatically removes the currently active document and makes the desired document the only active one.

In many respects the Aqua interface is reminiscent of earlier user interfaces for Macintoshes. 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 29) for more on this technology.

The Desktop and the Finder

A big part of the Aqua experience for users is the design of the desktop and the Finder, the primary interface for file-system interaction. Users are apt 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. It is an area at the bottom of the screen that 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 maximum resolution.

Preliminary © Apple Computer, Inc. May 2000

The new Finder for Mac OS X has a simple navigation interface that is fully contained within a single window. Large buttons instantly transport you to the most frequently accessed areas on your computer: your home directory, your applications, your documents, even to the people with whom you often communicate. The items that 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 new viewing mode called column view. This mode is ideal for navigating deep file systems; each click on 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 no longer brings up a separate window. Instead, the Finder replaces the old folder view within the single Finder window. 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.

Application Support

Part of the user experience is a near-seamless interaction among the various pieces of Mac OS X. This might be expected behavior for any operating system, but it is quite remarkable. 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's imposition of unity over this rich technological diversity is one of its singular accomplishments.

Instead of requiring users and developers to switch over abruptly to a brand new operating system, Apple has designed Mac OS X itself as a staging ground for a gradual transition. This is especially true in the area of application libraries and runtimes. Mac OS X supports three application environments, each intended for a particular type of application:

- The Classic environment lets you run all your Mac OS 9 applications. Because Classic is a compatibility environment, it does not support new OS X features, such as Aqua or core architectural enhancements provided by Darwin.

Preliminary © Apple Computer, Inc. May 2000

- The Carbon environment runs all Mac OS 8 and Mac OS 9 applications whose code has been optimized for Mac OS X. By converting their code to use the new 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 programming environments for creating the best next-generation applications.

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 HFS+ volume is copied to a UFS volume.

A new way of packaging applications 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 87) 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. Another user can log in to the same computer and have an entirely different set of preferences define his or her 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

Preliminary © Apple Computer, Inc. May 2000

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 on 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 package is just one type (see "Application Support" (page 19)). 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

Preliminary © Apple Computer, Inc. May 2000

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 "Application Packaging" (page 87) and "Bundles" (page 71) for further information.

Application Extensibility

Plug-ins are modules of code and resources that developers and users can dynamically add to an application to extend its capabilities. Mac OS X supports plug-ins with a new, generalized, system architecture. 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 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) become available, users can "plug" these features into the application.

For application developers, plug-ins yield a number of advantages. By providing a single, standard plug-in architecture, developers no longer have to design and implement their own 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.

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

Preliminary © Apple Computer, Inc. May 2000

Exported Application Services

Applications concurrently running 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. In addition to copy-paste and dragging operations, Mac OS X gives applications two mechanisms for sharing resources and capabilities: scripting and the Services menu.

Scripting in Mac OS X, as in Mac OS 8 and 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. Users can thus write scripts that link together the services of multiple applications in different environments.

The Services menu provides another avenue for applications to offer their capabilities to other applications. These "client" applications don't have to know what is offered in advance. How the Service menu works is simple. 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 she 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.

Preliminary © Apple Computer, Inc. May 2000

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 locally and on the Internet while her coffee is still warm. If users have questions, they can use Apple Help 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 2 for searching the Internet or an intranet as well as 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 foundation that is engineered for stability, reliability, and performance. This foundation is a core operating system commonly known as Darwin, which is also available as Open Source 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 a complete overview of Darwin, see the book *Inside Mac OS X: Kernel Environment*.

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 "under the cover"—typically, applications enjoy the benefits transparently. It manages processor resources such as CPU usage and memory, handles scheduling, enforces

Preliminary © Apple Computer, Inc. May 2000

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 some 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**. Guarantees low-latency access to processor resources for time-sensitive media applications.

Darwin also enables cooperative multitasking and preemptive and cooperative threading.

Preliminary © Apple Computer, Inc. May 2000

BSD

Integrated with Mach is a customized version of the BSD operating system (currently 4.4BSD). Darwin's implementation of BSD includes many of the POSIX APIs and exports these APIs 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
- kernel APIs

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 descriptions of the device drivers developed by Apple, see "Advanced Hardware Features" (page 37).

Preliminary © Apple Computer, Inc. May 2000

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 34).

File Systems

The file-system component of Darwin is based on an enhanced Virtual File System (VFS) design. VFS enables a layered architecture in which file systems are stackable. 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 on 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 on most Mac OS 8 systems and on Mac OS 9.
Mac OS Standard Format	Also called Hierarchical File System, or HFS. This is the volume format on 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.

Preliminary © Apple Computer, Inc. May 2000

Table 2-1 Supported local volume formats (continued)

UFS	Based on the 4.4BSD FFS - Fast File System. This is a "flat" disk volume format that is similar to the standard volume format of most UNIX operating systems and supports POSIX file system semantics 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 hard links and symbolic links. Although an alias and a symbolic link share are both light-weight references to a file or directory elsewhere in the file system—they are semantically different in significant ways. See the chapter "The Desktop and the File System" (page 119) for a description of these differences and how the Finder handles them.

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 principle file-sharing protocol on Mac OS 8 and Mac OS 9 systems.
NFS client	Network File Service, the dominant file-sharing protocol in the UNIX world.

Some file-system capabilities extend to all writable volume formats on Mac OS X. One of these is quotas; an administrator can specify disk quotas for all local and remote users of a system. Another global feature is file-system notification, a mechanism that automatically broadcasts notifications to all interested processes when a file is moved, deleted, or otherwise altered. The Desktop, Sherlock 2, and other system components make use of file-system notification.

Preliminary © Apple Computer, Inc. May 2000

Darwin and Open Source Development

Apple is the first major computer company to make open-source development a key part of its ongoing software 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 Environment*.

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 2D graphics and imaging capabilities of Mac OS X are based on Quartz, a new 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 new printing architecture and other graphics libraries such as QuickDraw and QuickTime.

Quartz

Quartz is a powerful new 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 uses PDF as an internal model for graphics representation.

Preliminary © Apple Computer, Inc. May 2000

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

Table 2-3 Quartz graphics capabilities

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 1024 pixels by 768 pixels as the minimum screen size for typical users. Resolutions of 640 x 480 and 800 x 600 are available for games and other multimedia applications.
Operators	Includes all standard PDF operators, including those for compositing and translucency.
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.
Velocity Engine	Quartz and QuickDraw both take advantage of the Velocity Engine to boost performance.
2D graphics acceleration	Supports two-dimensional graphics acceleration, improving what is currently available in QuickDraw.

Quartz has two components, Core Graphics Services and Core Graphics Rendering. The first of these, Core Graphics Services, 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 Core Graphics Rendering and QuickDraw. Core Graphics Services features such advanced capabilities as device-independent color and pixel depth, remote display, layered compositing, and buffered windows for the automatic repair of window damage.

The Core Graphics Rendering 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. Core Graphics Rendering uses PDF as an internal model for vector graphics representation. PDF offers several advantages, including good color management, internal compression, and font independence.

Preliminary © Apple Computer, Inc. May 2000

Core Graphics Rendering uses a coordinate system that is flexible and precise (because it uses floating-point coordinates) and thus permits some degree of device independence.

Core Graphics Rendering enables a number of important features:

- automatic PDF generation and save-as-PDF
- a consistent feature set for all printers
- automatic on-screen preview of graphics
- conversion of PDF data to printer raster data or PostScript
- high-quality screen rendering

See "The Graphics and Windowing Environment" (page 51) in the chapter "System Architecture" for more information on Quartz.

QuickDraw

For Carbon developers, QuickDraw is the primary library for the construction, manipulation, and display of two-dimensional graphical shapes, pictures, and text.

The Core Graphics Rendering library provides a QuickDraw (`grafPort`) programming interface for QuickDraw imaging instructions. This interface gives QuickDraw code access to the PDF-generation, PostScript-generation, 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 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 highly portable and the generated visual effects highly 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. Mac OS X's version of OpenGL produces consistently high-quality graphical images at a consistently high level of performance.

Preliminary © Apple Computer, Inc. May 2000

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 QuickTime 4. 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 on Mac OS 8, 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, and PNG. It also supports every significant professional file format for video, including AVI, AVR, DV, M-JPEG, MPEG-1, 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 has 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 movie "hot spots" and automatic Web-page launching.

Preliminary © Apple Computer, Inc. May 2000

Printing

The printing system for Mac OS X is based on a completely new architecture. 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 print dialogs. Table 2-4 describes some other 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 box.
PostScript printing	Prints to PostScript Level 1, 2, and 3 compatible printers (except in Classic environment).
Raster printers	Prints to raster printers in all environments, except in Classic.
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.
Print spooling	Enables speedy spooling of print jobs.

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

Preliminary © Apple Computer, Inc. May 2000

expensive operations such as synthesizing font data and rendering glyphs. ATS provides support for a wide variety of font formats including TrueType, Type 1, OpenType, and legacy bitmap fonts.

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.

Mac OS X's network protocol stack is based on BSD. The extensible architecture provided by Network Kernel Extensions, summarized in "Networking Extensions" (page 27), facilitates the creation of modules implementing new or existing protocols that can be added to this stack.

Media Types

Mac OS X supports the following network media types:

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.
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.

Preliminary © Apple Computer, Inc. May 2000

Standard Protocols

Mac OS X supports a number of protocols that are standard in the computing industry:

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).
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.
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.

Preliminary © Apple Computer, Inc. May 2000

Apple also implements a number of file-sharing protocols; see Table 2-2 (page 28) for a summary of these protocols.

Legacy Network Services and Protocols

Apple is including a number of its legacy network products in Mac OS X. This will ease the transition to the new operating system for Mac OS users who currently depend on these products.

- AppleTalk is a suite of network protocols that is standard on Macintosh and can be integrated with other network systems, such as the Internet. The protocols include ATP/ASP (Apple Transaction Protocol/Apple Session Protocol) and ADSP, a transmission layer protocol. 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.

Routing and Multihoming

Mac OS X is a powerful and easy-to-use desktop operating system but will also serve as the basis for powerful server solutions. Some businesses or organizations have small networks that could benefit from the services of a router and want to leverage their general purpose server. 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) and OSPF (Open Shortest Path First) are used in the implementation of this feature.

Mac OS X also allows multihoming using IP aliasing. 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.

Preliminary © Apple Computer, Inc. May 2000

Personal File and Web Services

Personal Web Sharing, which is also a feature of Mac OS 8 and 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 34), "File Systems" (page 27), and "Networking Extensions" (page 27). 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

Preliminary © Apple Computer, Inc. May 2000

- mass storage (Zip and Jaz drives, for instance, and external hard drives)
- imaging
- display
- audio

FireWire

FireWire is Apple's implementation of the new 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, digital video disks (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.

Preliminary © Apple Computer, Inc. May 2000

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 41)). 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 45) 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 51) and "Other Application Services" (page 61) 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 62) for more information.

Preliminary © Apple Computer, Inc. May 2000

- **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 40) and the book *Inside Mac OS X: Kernel Environment*.

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 111)). Many public APIs of the kernel environment are exported through the System framework.

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.

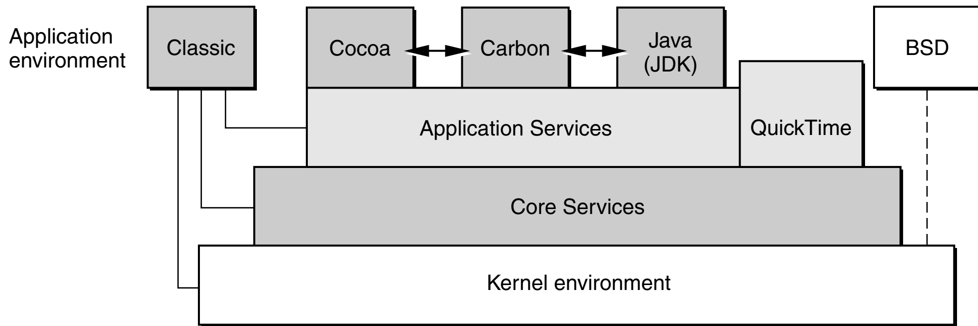
A Layered Perspective

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.

Preliminary © Apple Computer, Inc. May 2000

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 (page 41) represent the different application (or execution) environments of Mac OS X. There are five such environments. The first two are the Classic and the BSD Commands environments.

- The Classic "compatibility" environment is where users can run their non-Carbon 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 that developers can specifically compile code for on 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.
- 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

Preliminary © Apple Computer, Inc. May 2000

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.

In developer versions of Mac OS X, the kernel environment exports BSD services and commands to the upper layers of the system through the System framework. User versions of Mac OS X do not. Because the BSD command environment is a special optional environment, it is not described further in this document.

Next 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 45) 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 47) 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 49) 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 118) in the chapter "Umbrella Frameworks.")

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 51) and "Other Application Services" (page 61) 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 41) presents QuickTime as straddling the line

Preliminary © Apple Computer, Inc. May 2000

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 62) 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
- Virtual File System–based file systems
- 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.

Preliminary © Apple Computer, Inc. May 2000

- **BSD**. A version of 4.4BSD that is used to support 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 Embedded 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 15)).
- **Networking**. Most 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. The kernel environment's native networking protocols and facilities are described in "Networking and the Internet" (page 34) in the chapter "System Technologies."
- **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. By using the Virtual File System (VFS) infrastructure, developers can write kernel extensions that add support for other file systems and extend file system functionality—adding file-level compression, for instance. 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 27) in the chapter "System Technologies."

As described in "Darwin and Open Source Development" (page 29) of the chapter "System Technologies," the kernel environment is a subset of Darwin, Apple's Open Source technology. Darwin combines the Mac OS X kernel environment and

Preliminary © Apple Computer, Inc. May 2000

the BSD commands and libraries essential to the BSD Commands environment. For more on the Mac OS X kernel environment and its relation to the Darwin, see the book *Inside Mac OS X: Kernel Environment*.

The kernel environment, Core Services, and Application Services layers of Mac OS X are packaged as **umbrella frameworks**. The major Mac OS X application environments, Carbon and Cocoa, are also packaged as umbrella frameworks. See the chapter "Umbrella Frameworks" (page 111) 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.

Preliminary © Apple Computer, Inc. May 2000

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 Mac OS 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.

New Managers. Apple has developed new Carbon versions of the Printing Manager and the Event Manager for Mac OS X. The old Printing Manager is not supported and developers must use the Carbon Printing Manager. The old Event Manager is still supported; however, developers are strongly encouraged to adopt the Carbon Event Manager.

Replacement Managers. Different Carbon technologies now take the place of earlier libraries.

Table 3-1 Carbon Managers

Instead Of	Now Use
AppleTalk Manager	Open Transport
PPC Toolbox	Apple events
Standard File Package	Navigation Services
QuickDraw 3D	OpenGL
Help Manager	Help Viewer

Developers must use these replacements.

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

Preliminary © Apple Computer, Inc. May 2000

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 programs based on 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 62)), 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 *CarbonLib 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.

Preliminary © Apple Computer, Inc. May 2000

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, dialog boxes, buttons, tables, text fields, sliders, pop-up lists, scroll views, 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 dialog boxes 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.

Preliminary © Apple Computer, Inc. May 2000

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.

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, the latest 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 machine 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 47). 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 on 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. 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 the Java virtual machine, the "just-in-time" (JIT) bytecode compiler, and the basic Java packages.

Preliminary © Apple Computer, Inc. May 2000

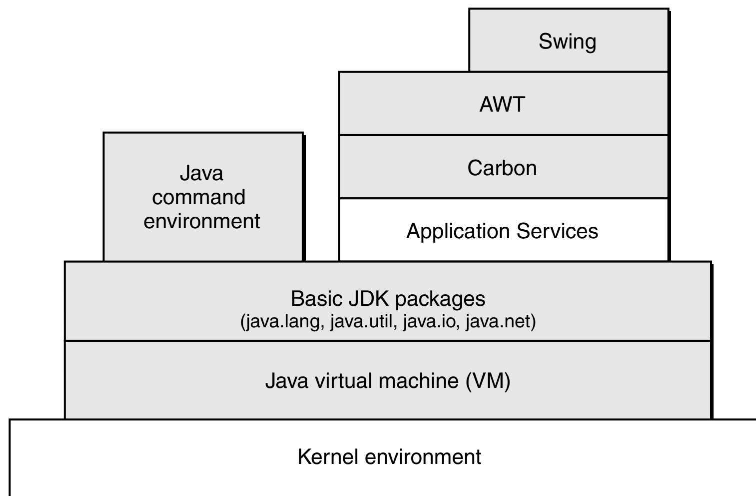
The Java virtual machine is located at `/System/Library/Frameworks/JavaVM.framework/Libraries`. The basic packages include `java.lang`, `java.util`, `java.io`, and `java.net`; they are in the `classes.jar` archive in the `Classes` folder of the same framework.

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

  The more significant of these packages are `java.awt` and `java.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`.

The architecture of the Java application environment is much different, and more complex, than the simplified picture in Figure 3-1 (page 41) indicates. Figure 3-2 presents a more realistic view of the Java environment.

Figure 3-2 Architecture of the Java environment



Preliminary © Apple Computer, Inc. May 2000

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 on 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 the Human Interface Toolbox or the Application Kit.

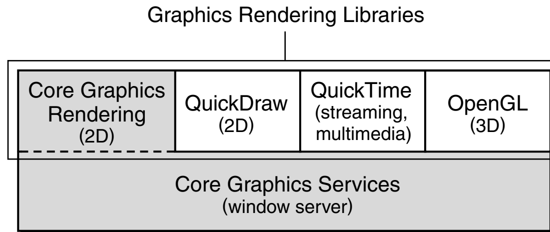
The Graphics and Windowing Environment

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, Core Graphics Services and Core Graphics Rendering.

Preliminary © Apple Computer, Inc. May 2000

Figure 3-3 Mac OS X graphics and windowing environment



The Core Graphics Rendering 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 Core Graphics Rendering, QuickDraw, and OpenGL.

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

Core Graphics Services consists of the Mac OS X window server and the (currently private) system programming interfaces (SPIs) it depends on. 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.

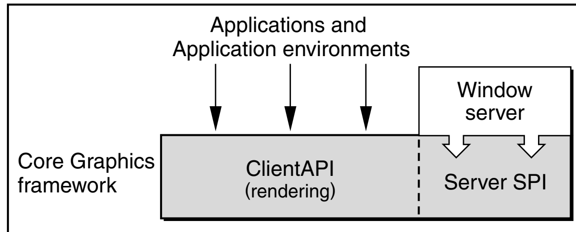
Quartz is largely implemented in the Core Graphics framework (`CoreGraphics.framework`). The dynamic shared library of this framework, as illustrated by Figure 3-4, includes both client APIs and server SPIs. Applications or

Preliminary © Apple Computer, Inc. May 2000

application environments link with the client side of the library—Core Graphics Rendering—for screen rendering, PDF generation, and other services. The major client of the server SPIs of the Core Graphics framework is the window server itself.

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



Core Graphics Services

The Core Graphics Services layer of Mac OS X comprises the window server and the (private) system programming interfaces (SPI) on which the window server depends. In this layer are the facilities responsible for rudimentary screen displays, window composition 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.

Preliminary © Apple Computer, Inc. May 2000

One of the primary duties of the window server is window composition. It composites and recomposites each pixel of an application's window as the window is drawn, redrawn, covered, and uncovered. Windows are represented as a bitmap that includes both translucency (alpha channel) and anti-aliasing information. The bitmap is buffered, allowing the window server to "remember" an application's window and to recomposite it without the application's involvement. However, Quartz does not retain vector information that a graphics library (such as its own Core Graphics Rendering) might have used to create a window or any other image.

In its Core Graphics Services 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 67).

Core Graphics Rendering

The Core Graphics Rendering 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.

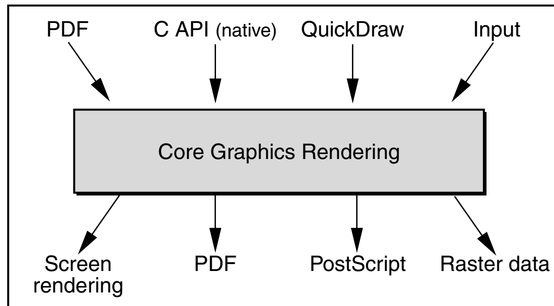
By using vectors, Core Graphics Rendering can also use a coordinate system for drawing based on, say, inches or centimeters rather than a pixel grid. The coordinate system is flexible, permitting various measurement standards, and it enables some degree of display independence since it is not bound to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

Preliminary © Apple Computer, Inc. May 2000

The internal model that Core Graphics Rendering uses for vector graphics representation is Portable Document Format (PDF). As a superset of Adobe PostScript, PDF brings several improvements, including better color management, internal compression, font independence, and interactivity. However, PDF is not a full-fledged language as is PostScript; it is declaratively, not programmatically, specified. Consequently a sophisticated and expensive language runtime is not necessary, as it is for PostScript.

You can think of Core Graphics Rendering as a "black box" that converts input to PDF and then converts the PDF to various output formats. Figure 3-5 illustrates this.

Figure 3-5 Core Graphics Rendering inputs and outputs



The primary inputs for Core Graphics Rendering are the drawing commands and statements made with QuickDraw and the native C APIs. (Future APIs in the front end may be supported.) These commands and statements 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.

Core Graphics Rendering, as the foregoing paragraph suggests, is the underlying engine for the Mac OS X printing system. Printing is often a two-pass affair. Core Graphics Rendering 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 Core Graphics Rendering again to convert it to the appropriate output format.

Preliminary © Apple Computer, Inc. May 2000

The Printing System

The Mac OS X printing system provides a flexible and powerful new printing environment for Macintosh users. The new printing system presents a refined user interface that makes setting up a printer easy and intuitive for the average user, yet it also has the necessary features to support the requirements of advanced users and administrators. The printing system's modular, client/server architecture

- makes it much easier for printer vendors to write Macintosh drivers and extend printing dialog boxes
- uses PDF-based rendering, providing PDF capability for all printers, including inexpensive raster printers
- 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
- provides applications and printer drivers control over individual user interface elements in the system's printing dialog boxes, obviating the need to completely replace the standard Print or Page Setup dialog boxes with custom versions

A key aspect of the new printing system's design is its robust support for Carbon applications. Because the Carbon Printing Manager is supported on 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 on Mac OS 8 and 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 dialog boxes and flexible printing architecture.

User Interface

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, Internet-based print jobs.

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

- **PrintCenter**. Allows the user to locate, select, and configure available printers, and to determine the status of print jobs associated with each.

Preliminary © Apple Computer, Inc. May 2000

- **Page Setup dialog box**. Allows the user to specify the format of the document to be printed.
- **Print dialog box**. Allows the user to specify the parameters of a print job, and to print a document on a specified printer.

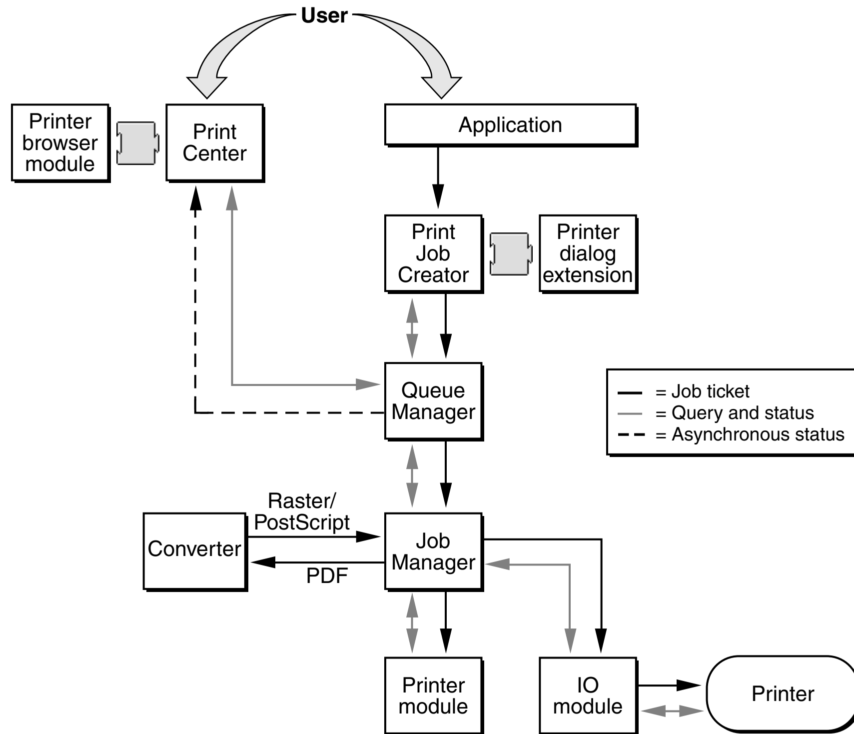
The new printing system's interface includes a number of important improvements in both ease-of-use and stability relative to the Mac OS 8 and 9 printing model. The Chooser—the most common source of user confusion when dealing with printers—is replaced by PrintCenter, which combines many of the features of the Chooser and desktop printing into a single, integrated interface. Unlike the Chooser, PrintCenter 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 dialog boxes are standardized for all printers, and are easily extendable to allow for third-party customization. The Print dialog box allows users to choose a variety of destinations—printers, files, and fax, for example—using a convenient pop-up menu.

Architecture Summary

The Mac OS X printing architecture consists of a set of nine modules. Conceptually, these modules can be divided into client and server groupings. Four of the modules—PrintCenter (with optional printer browser modules) and the Print Job Creator (with optional printing dialog extensions)—make up the client side. These modules are responsible for presenting all user interface elements, accepting the raw drawing commands from applications, and passing the data to be printed on to the print server. The remaining five modules—Queue Manager, Job Manager (with optional 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. These modules and their relationships are depicted in Figure 3-6.

Preliminary © Apple Computer, Inc. May 2000

Figure 3-6 Mac OS X printing system



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

- **Print Job Creator (PJC)** Implements the Carbon Printing Manager API used by applications. Displays the Print and Page Setup dialog boxes, captures drawing information from applications, and passes the data to a local or remote Queue Manager for printing.
- **Printing dialog extensions (PDEs)**. Extends a Print or Page Setup dialog box, allowing third parties to add user interface elements in support of specific printers. A PDE is paired with a printer module which interprets and applies the custom settings offered by the PDE.
- **PrintCenter**. Allows the user to locate and select printers, as well as control and obtain status for print jobs.

Preliminary © Apple Computer, Inc. May 2000

- **Printer browser module (PBM)**. Extends the PrintCenter 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.
- **Queue Manager**. Handles the queuing of print jobs after they leave the Print Job Creator. Responds to requests from PrintCenter to manipulate or return status information about print jobs in the queue. Reports errors back to the PrintCenter, or directly to the user if PrintCenter is not active.
- **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**. Assists the 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 user choices to control the printing of the job. Job tickets are created by the Print Job Creator and are updated by each component at every step in the printing process.

Printer Discovery

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

During printer discovery, the printing system compiles one list of available printers per connection (transport) type. First, the Queue Manager enumerates all of the known I/O modules, the connection types they support, and the printer browser modules they are associated with. This information is passed to PrintCenter which populates the connection pop-up menu with the connection types returned. When the user selects a connection type, PrintCenter passes the requested connection type to Queue Manager which then asks all the printer modules (through the Job Manager) if they support it.

Preliminary © Apple Computer, Inc. May 2000

Once the Queue Manager knows which printer modules support the requested connection type, it then asks those printer modules for information about printer models and languages supported on that connection. Queue Manager passes this information (along with any custom icons) back to PrintCenter, which in turn passes it to the appropriate printer browser modules. The PBMs use the printer type information to browse the selected connection type for printers. If a match is found, the printer item is displayed in the browser window along with the icon and language information.

When the user clicks a printer to add it to the workset, PrintCenter then gets the selected printer address, icon, and printer model information from the PBM. PrintCenter uses this information to create a new print queue and add it to the workset.

The Printing Process

Before a document is printed, the user brings up the Page Setup dialog box so she can define the document's format. The application accomplishes this by calling into the Print Job Creator (PJC). An optional printing dialog extension (PDE) hosted by the PJC may extend the Page Setup dialog box to include options specific to the application's drawing environment—custom page layouts, for example. After page setup is complete, the user requests that the application display (again using the PJC) the Print dialog box, with which they define the parameters of the print job. As with the Page Setup dialog box, the Print dialog box may include application- and/or printer-specific options added by a PDE.

When the user dismisses the Print dialog box, the Print Job Creator accepts drawing commands from the application (QuickDraw, Core Graphics, or a PDF file) and passes the data to the Queue Manager along with a job ticket describing the printing parameters. The Queue Manager then passes the data and job ticket to the Job Manager, which is responsible for managing the rest of the printing process. Once the job is sent to the Queue Manager, all errors associated with the job are reported asynchronously back to the PrintCenter.

The 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 Job Manager uses a converter to transform the incoming data into a format that the destination printer module can accept. Next, the 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

Preliminary © Apple Computer, Inc. May 2000

render the data. Finally the 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 145) in the chapter "Inter-Environment Issues."

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 67).

Preliminary © Apple Computer, Inc. May 2000

The Pasteboard

The pasteboard is a background server, similar to the Mac OS 9 Clipboard, that allows you to transfer data between applications. The term "pasteboard" is preferred to "clipboard" because the former connotes that the pasteboard can hold multiple representations of the same data. The pasteboard 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.

Core Services

The Core Services layer contains the system services that do not have any effect on application's graphical user interfaces. Core Services is associated with the Core Services umbrella framework (`CoreServices.framework`).

Carbon Managers

The Core Services layer includes a number of Carbon managers that offer low-level services to all application environments. These services include cooperative and preemptive threading, resource management, memory management, and file-system operations. Table 3-2 summarizes some of these managers.

Table 3-2 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.

Preliminary © Apple Computer, Inc. May 2000

Table 3-2 Carbon managers in the Core Services layer (continued)

Manager	Description
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, CD-ROMs, 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.
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.

Preliminary © Apple Computer, Inc. May 2000

Table 3-2 Carbon managers in the Core Services layer (continued)

Manager	Description
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.

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

Preliminary © Apple Computer, Inc. May 2000

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 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.
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.
Bundle Services	CFBundle	Offers an elegant means of organizing and locating many types of program resources including images, sounds, localized strings, and executable code.
Plug-in Services	CFPlugIn	Provides a standard plug-in architecture for Mac OS X applications (as well as Mac OS 9 applications).
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.
URL Services	CFURL CFURLAccess	Gives programs a way to access, via URLs, resources stored locally or remotely.

Preliminary © Apple Computer, Inc. May 2000

Table 3-3 Core Foundation services (continued)

Services	Types	Description
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.
Preferences Services	CFPreference	Enables programs to store and retrieve user preferences. See "The Preferences System" (page 141) in chapter "The Preferences System" for background information.
XML Parser	CFXMLParser	Provides a nonvalidating XML parser for reading and extracting data from XML documents.
Notification Services	CFNotificationCenter,	Implements distributed notifications, a mechanism that allows a process to send messages (notifications) to other processes on the same machine.
Run Loop Services	CFSocket, CFRunLoop (and related)	Provides low-level event-handling and dispatch services.
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.

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 on 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.

Preliminary © Apple Computer, Inc. May 2000

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 149) in the chapter "Inter-Environment Issues" for further discussion of Apple events.

Open Transport

Open Transport is the primary user-level networking and communications software for Mac OS X. It 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 on 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

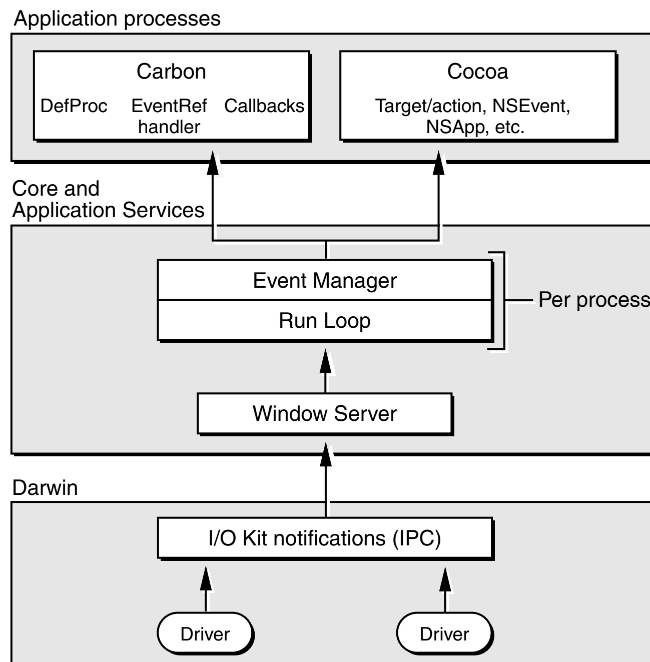
The perspective that Figure 3-1 (page 41) gives of Mac OS X as layers of system software suffices to illustrate the general interfaces and dependencies among parts of the system. But it does not adequately convey the dynamism of the operating system—in other words, how Mac OS X typically "works." An 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,

Preliminary © Apple Computer, Inc. May 2000

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 box).

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



A low-level event originates when the device driver that controls an input device such as a mouse or the keyboard detects a user action. The I/O Kit, which forms the foundation of all device drivers on Mac OS X, creates the event and puts it in the window server's event queue (see "Core Graphics Services" (page 53) 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).

Preliminary © Apple Computer, Inc. May 2000

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, an opaque low-level event structure. 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 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.

Preliminary © Apple Computer, Inc. May 2000

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 system language preferences.

Preliminary © Apple Computer, Inc. May 2000

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 97) for more information on these types of bundles.

Benefits of Using Bundles

Bundles provide a variety of important advantages over the traditional Mac OS 8 software packaging scheme.

- A single bundle executable can run on Mac OS 8, 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 AltiVec).
- 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 and dropping them.

Preliminary © Apple Computer, Inc. May 2000

Versioned bundles (described in “Anatomy of a Bundle” (page 73)) 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, the “new-style” bundle 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
```


Preliminary © Apple Computer, Inc. May 2000

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 following structure:

Listing 4-1 A minimal bundle
(Legend: * = file; - = opened directory; + = closed directory)

```
- MyBundle
    - Contents
        * PkgInfo
        * Info.plist
```

In other words, the `Contents` directory and, inside it, the `PkgInfo` and `Info.plist` files 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.

The information property list, `Info.plist`, contains key-value pairs stored in XML format. These pairs specify attributes such as the name of the main executable for the bundle, version information, type and creator codes, and other meta data. 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 132) in the chapter Software Configuration for more on information property lists, including an example of one.

The other special bundle file is called `PkgInfo`. This file contains only the type and creator codes for the bundle. Although the information is redundant—it is kept in the information property list as well—the `PkgInfo` file acts as a cache that improves performance for applications such as the Finder that need quick access to the type and creator codes for files. See the chapter "The Desktop and the File System" (page 119) for more information on how Finder processes the information in the `PkgInfo` and `Info.plist` files.

Preliminary © Apple Computer, Inc. May 2000

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.

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. The following example shows what might go into such a bundle.

Listing 4-2 The bundle layout of a complex application
(Legend: * = file; - = opened directory; + = closed directory)

```
- MyBundle
    * MyApp (alias to Contents/MacOSClassic/MyApp)
    - Contents
        - MacOSClassic
            MyApp
            Helper Tool
        - MacOS
            MyApp
            Helper Tool
        * Info.plist
        * PkgInfo
        - Resources
            * Hand.tiff
            * Horse.jpg
            + WaterSounds
            - en_US.lproj
                * bird.tiff
                * Bye.txt
                * house.jpg
                * house-macos.jpg
                * house-macosclassic.jpg
                * InfoPlist.strings
                * Localizable.strings
                + CitySounds
            - en_GB.lproj
                * bird.tiff
                * Bye.txt
                * house.jpg
                * house-macos.jpg
```


Preliminary © Apple Computer, Inc. May 2000

```
          * house-macosclassic.jpg
          * InfoPlist.strings
          * Localizable.strings
          + CitySounds
      - Japanese.lproj
          * bird.tiff
          * Bye.txt
          * house.jpg
          * house-macos.jpg
          * house-macosclassic.jpg
          * InfoPlist.strings
          * Localizable.strings
          + CitySounds
  + Frameworks
  + Plug-ins
  + SharedFrameworks
  + 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`, `SharedFrameworks`, `SharedSupport` and `Plug-ins` directories are optional and appear only as necessary.

Several directories contain, as their names suggests, 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" means "language"). Each such directory contains all localizable resources for a particular language and often 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. See "Localized Resources" (page 81) for more on bundle resources.

Preliminary © Apple Computer, Inc. May 2000

A bundle almost always stores each resource in its own file instead of grouping them in a single file, as does the Mac OS 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.

Important
You should 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 `SharedFrameworks` directory contains frameworks that are also part of the application package; but the versions of these frameworks will be checked against the system registry to see if there are more recent versions available. If a more recent version is found in the system, the version in the `SharedFrameworks` directory is ignored. The inclusion of versioned frameworks in the application package makes it possible for an application to be completely self-contained. An application can be installed, relocated, and removed with a simple drag and drop.

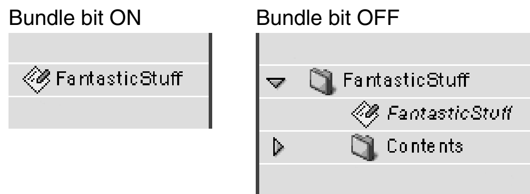
The `Frameworks`, `SharedFrameworks`, `Plug-ins`, and `SharedSupport` directories occur mostly in application bundles. See the chapter "Application Packaging" (page 87) for further information on these directories.

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 4-1 (page 78)). 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.

Preliminary © Apple Computer, Inc. May 2000

Figure 4-1 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 78)), 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 Desktop and the File System" (page 119).

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.

Preliminary © Apple Computer, Inc. May 2000

- **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 the system bundle routines recognize and handle appropriately (although their internal structure is different from other loadable bundles). These bundles have an extension of `.kext`. The Kernel Manager, which claims KEXTs as a document type, dynamically loads them into the kernel environment.

Bundles must have an extension appropriate to their type. For applications, that extension is `.app`. For the developmental variants of applications, the extensions should be `.debug` and `.profile`. The 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`. The Finder does not show the `.app` extension.

An Application's Main Bundle

With the exception of some 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.

Preliminary © Apple Computer, Inc. May 2000

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. 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 Finder) and will launch your application when the user double-clicks them.

Preliminary © Apple Computer, Inc. May 2000

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 73) 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 82) for details on the exact process Mac OS X uses to locate a bundle resource.

Preliminary © Apple Computer, Inc. May 2000

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. It also provides 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. Consult the development-environment documentation for details

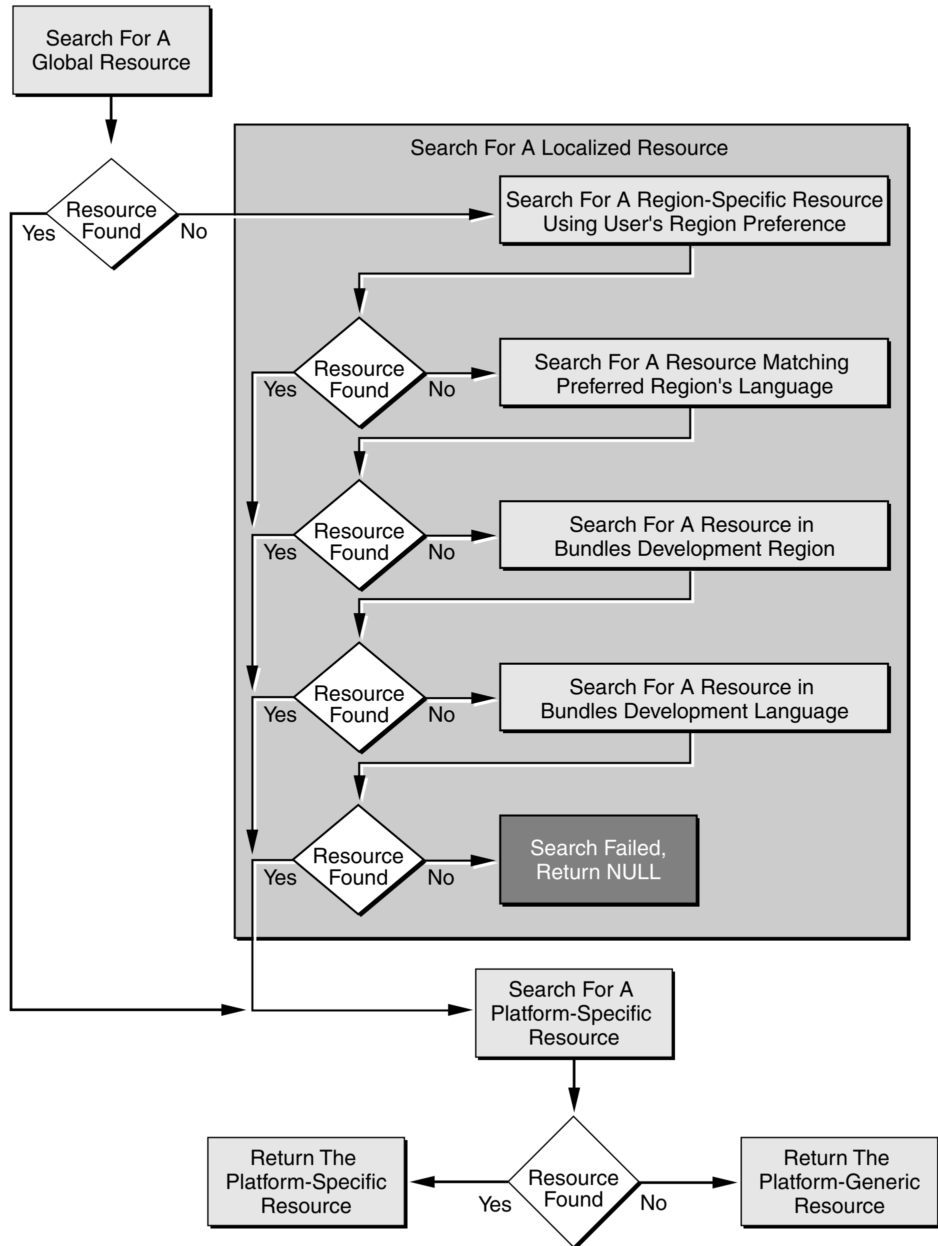
Search Algorithm

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 groping around inside the bundle yourself.

The Figure 4-2 details the steps a system routine uses to locate a resource.

Preliminary © Apple Computer, Inc. May 2000

Figure 4-2 Locating a resource in a bundle



Preliminary © Apple Computer, Inc. May 2000

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 8) 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 Mac OS 8–specific name would be `Fish-macosclassic.jpg`. When an application running on Mac OS 8 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 `.rsrc` files, which are treated as bundle resources just like any other kind of file. It is possible to use the CFBundle API to get a CFURL to such a file, convert that to an FSRef, and then open it using the Resource Manager. There are, however, two special resource files that CFBundle will manage for you if you provide them. One is for non-localized 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`.

Preliminary © Apple Computer, Inc. May 2000

This file is stored in the appropriate `.lproj` directory, one version for each language or region. Note that the resources should be 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 CFBundle function provided specifically for this purpose.

Preliminary © Apple Computer, Inc. May 2000

C H A P T E R 5

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 8, 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, wizards, 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 71).

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 71), is a directory containing executable code and the resources to support that code. Application bundles as well as

Preliminary © Apple Computer, Inc. May 2000

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 package can run, without modification, on Mac OS 8, Mac OS 9, and Mac OS X.
- Applications can include different localizations. Applications can automatically display the set of localized resources 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 may run applications on a server.
- Customers can easily download applications from a website or email them.
- 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.

Preliminary © Apple Computer, Inc. May 2000

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 71). For further information about the Desktop, the Finder, and their roles in relation to applications, see the chapter "The Desktop and the File System" (page 119).

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 packages 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 the application links with. 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; this includes frameworks that are shared (see "Shared Frameworks and the Central Directory" (page 91) for the handling of shared frameworks).

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

Preliminary © Apple Computer, Inc. May 2000

```
Frameworks/
SharedFrameworks/
SharedSupport/
Plug-ins/
```

The remainder of this section explains the purposes and issues related to the first three of these directories. For a description of the `Plug-ins` directory, see "Applications and Loadable Bundles" (page 93).

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` whereas it may or may not use the code in `SharedFrameworks`. If a framework or shared library is missing from `Frameworks`, the application cannot launch.

Listing 5-1 illustrates how a typical private framework might be stored in the application bundle.

Listing 5-1 Location of an application's private framework

```
FantasticApp.app/
    Contents/
        PkgInfo
        Info-macos.plist
        MacOS/
        Resources/
        Frameworks/
            GoodStuff.framework/
        SharedFrameworks/
        SharedSupport/
        Plug-ins/
```


Preliminary © Apple Computer, Inc. May 2000

Shared Frameworks and the Central Directory

The `SharedFrameworks` directory contains frameworks that are also part of the application package, but these frameworks are meant to be shared with other applications. Shared frameworks of an application are guaranteed to be forward compatible, whereas frameworks private to an application don't have to be.

To facilitate sharing of the most recent version of code in `SharedFrameworks`, Mac OS X uses a central directory, or registry. This central directory tracks the versions of shared frameworks and other shared software in all installed application packages. Before an application dynamically loads framework code, the system checks the version of the required framework in `SharedFrameworks` against the central directory to see if more a recent version of the same framework is available. If a more recent version exists, the framework in the `SharedFrameworks` directory is ignored and the one identified by the central directory is used. If no corresponding framework is found in the central directory, or if the version of the framework it is earlier, the framework in the application's `SharedFrameworks` directory is used.

The inclusion of shared frameworks and other shareable software in the application package contributes to application self-sufficiency. To install, relocate, or remove an application, users simply drag the application icon and drop it the appropriate place. An application so installed might not use the most recent version of a shared framework, but at least it should be able to execute with the frameworks packaged with it. By keeping track of versioned frameworks within all application packages, the central directory ensures that an application remains "read-only" and that pieces of it are not duplicated all over a system. At the same time, the central directory makes it possible for related applications to use the latest shared frameworks installed on a system.

Listing 5-2 shows where shared frameworks go in an application package.

Listing 5-2 Location of an application's shared framework

```
FantasticApp/
    Contents/
        PkgInfo
        Info-macos.plist
        MacOS/
        Resources/
```


Preliminary © Apple Computer, Inc. May 2000

```
        Frameworks/
        SharedFrameworks/
            GreatStuff.framework/
        SharedSupport/
        Plug-ins/
```

Other Shared Application Code

Any supplementary, shareable application code that is not a framework, shared library, or loadable bundle (including plug-ins) goes in the `SharedSupport` directory of the application package. Examples of this class of code are helper applications, assistants, and tools. As with shared frameworks, the latest versions of software in this directory is shared among related applications using the central-directory mechanism.

In the example in Listing 5-3 (page 92), FantasticSpreadsheet, which is part of an office-productivity suite of applications, includes a small graphing application in `SharedSupport`. The FantasticSpreadsheet application and its sibling application, FantasticDatabase, jointly use FantasticGrapher.

When FantasticSpreadsheet is installed, the version of FantasticGrapher is recorded in the central directory. When the user attempts to run FantasticGrapher, the system checks the version of the helper application in `SharedSupport` against the latest version of the same application in the central directory. It runs whichever version is most recent.

Listing 5-3 Location of an application's shared code (nonframework)

```
FantasticSpreadsheet/
    Contents/
        PkgInfo
        Info-macos.plist
        MacOS/
        Resources/
        Frameworks/
        SharedFrameworks/
        SharedSupport/
            FantasticGrapher
        Plug-ins/
```


Preliminary © Apple Computer, Inc. May 2000

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, shared 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 97) 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 `Plug-ins` 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` folder 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` folder 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).

Preliminary © Apple Computer, Inc. May 2000

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. See the folowing sections for information on the preferred locations.

Application Help

On Mac OS X, the Help Viewer application displays help information for both applications and 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 can package the files in a help book (the standard format for Mac OS 9) or in a help bundle—which has the same internal organization as any bundle and that takes the extension of `.help`. The files (plus associated images) in help bundles or help books can be localized for multiple languages and regional dialects. The text files must be HTML 3.2-compliant and otherwise conform to the specification for Apple Help books. See the Apple Help SDK documentation for instructions on creating help bundles and preparing and indexing help files.

Note: Although help bundles are the preferred packaging format in Mac OS X—because of the internationalization infrastructure they afford—Help Viewer can also understand Mac OS 9 help books.

The information property list (`Info.plist`) of an application should contain a key, CFBundleHelpBookFolder, whose value specifies the name of a directory in the Application's `Resources` directory that contains help. If this directory is a `.help`

bundle, it goes in the top-level location of `Resources` reserved for nonlocalized resources; it belongs in the nonlocalized location since bundles contain their own localized resources. If help is in the form of a help book, one or more localized books go in the appropriate `.lproj` directories of `Resources`; the name of each book directory must be the same—that is, they should not be localized.

When a user chooses the Help menu command for the application (for example, MyApp Help), the application can call system functions that might use the CFBundleHelpBookFolder key to locate the version of help corresponding to the user's language preference. It then requests the Help Viewer to display the help. The Apple Help APIs give applications other 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.

If you package help in a help bundle, the `Info.plist` file for the bundle must contain a CFBundleHelpTOC key-value pair. The value specifies the HTML file in the bundle that contains the special meta tags required by Apple Help. This file is typically the "entry point" or first page of the help.

You can also put application help as well as more general help, in both book and bundle form, outside the application package. Such help should go in one of the standard locations for third-party help: `/Network/Documentation/Help` or `Library/Help` in the 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. However, application help external to the application is discouraged because it runs counter to the "all in one place" model of the application package. If your application help is in the form of a help book, and you have it 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.

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 141) in the chapter "Software Configuration").

Your application should never write user-preference data inside the application package. Preferences are stored in the `Library/Preferences` folder of the logged-in user's account (local or network) or in the same location in the machine-local or

Preliminary © Apple Computer, Inc. May 2000

network domains. You should never write preferences data directly to these locations; instead use the APIs offered by Core Foundation's 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 wizards. 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 93)):

- 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 location 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.

Preliminary © Apple Computer, Inc. May 2000

Frameworks

A framework is a type of bundle that packages a dynamic shared library with the resources that the library requires, including header files and reference documentation. This consolidation of code and resources brings with it a number of benefits. For example, it makes it easier for the 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 point 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

Preliminary © Apple Computer, Inc. May 2000

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 73) in the Bundles chapter 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 99).

Preliminary © Apple Computer, Inc. May 2000

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 on 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:

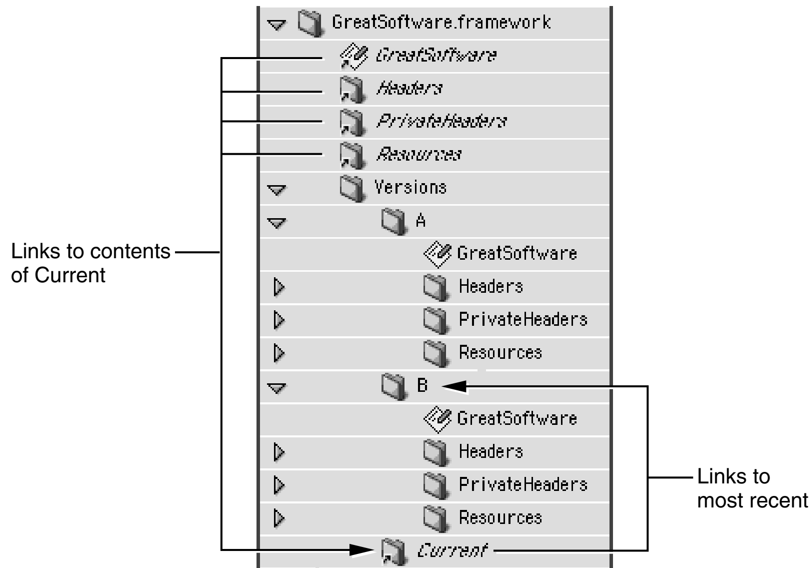
```
GreatSoftware.framework/
    GreatSoftware
    Headers/
    PrivateHeaders/
    Resources/
    Versions/
```

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 6-1 illustrates how this linking is done.

Preliminary © Apple Computer, Inc. May 2000

Figure 6-1 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 104).

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`. More specifically, 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 underbar 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

Preliminary © Apple Computer, Inc. May 2000

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 non-localized 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

Important
Much of the information in this section will be out of date as soon as frameworks are converted to use the new bundle structure and the new file-system layout.

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 /Local/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 the given order:

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


Preliminary © Apple Computer, Inc. May 2000

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.

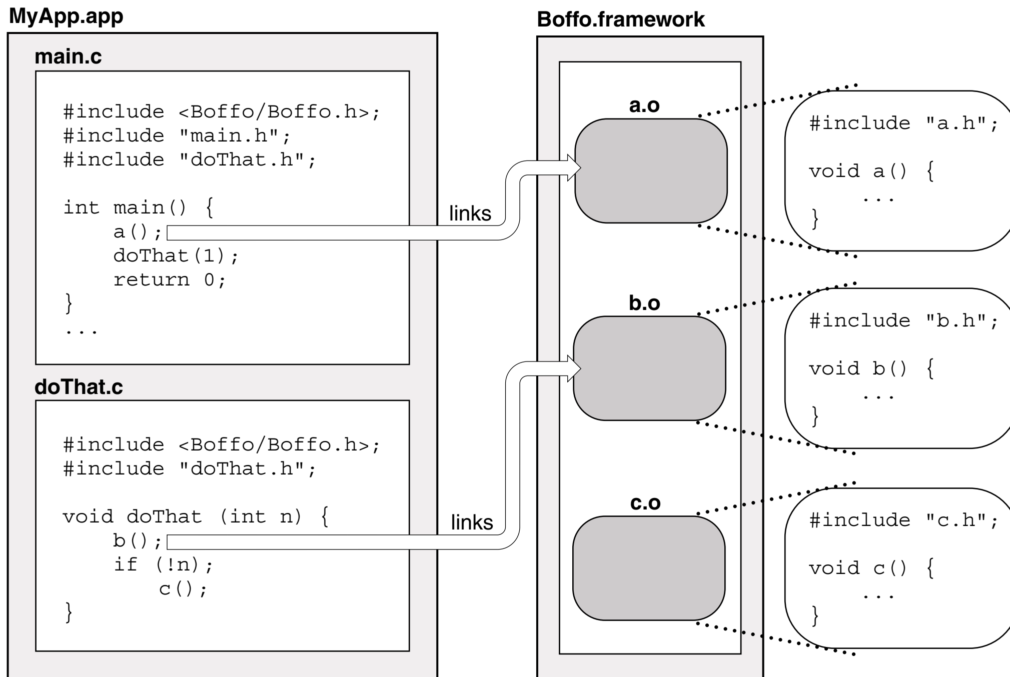
Dynamic shared libraries have characteristics that set them apart from statically linked shared libraries. With dynamic shared libraries, the binding of undefined symbols in a program is delayed until the execution of that program. In other words, the **dynamic link editor** not only attempts to resolve 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 6-2 illustrates this “lazy linking” behavior. In this example, module `a.o` is linked in

Preliminary © Apple Computer, Inc. May 2000

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 6-2 Lazy linking of dynamic shared library modules



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.

Preliminary © Apple Computer, Inc. May 2000

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.

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 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 104) 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.

Preliminary © Apple Computer, Inc. May 2000

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 99)). 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

Preliminary © Apple Computer, Inc. May 2000

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 99) 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 were 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).

Preliminary © Apple Computer, Inc. May 2000

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).

Preliminary © Apple Computer, Inc. May 2000

Table 6-1 (page 108) summarizes the salient facts about each type of version.

Table 6-1 Summary of framework versioning

Type of version	When required	What happens
Major/incompatible (backward compatible)	API changes (such as renaming functions); removing API; adding or reordering instance variables; adding 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)	Adding 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.

Guidelines for Major Versioning

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:

Preliminary © Apple Computer, Inc. May 2000

- 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.)

If You Do, Don't Clean The Project

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.

Preliminary © Apple Computer, Inc. May 2000

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 87) 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.

Preliminary © Apple Computer, Inc. May 2000

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 97) describes frameworks in detail. A frameworks 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 71) for detailed information on bundles.

Second, frameworks can be one of several types, or "flavors." To begin with, frameworks are 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 user's `Library` folder; however, if frameworks are closely bound to an application, they typically go inside the application package (see "Application Packaging" (page 87)). Public frameworks are shipped to customers and their APIs are exposed through their header files. By convention, they are installed in the `Frameworks` folder 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 folder 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.

Preliminary © Apple Computer, Inc. May 2000

The Purpose of Umbrella Frameworks

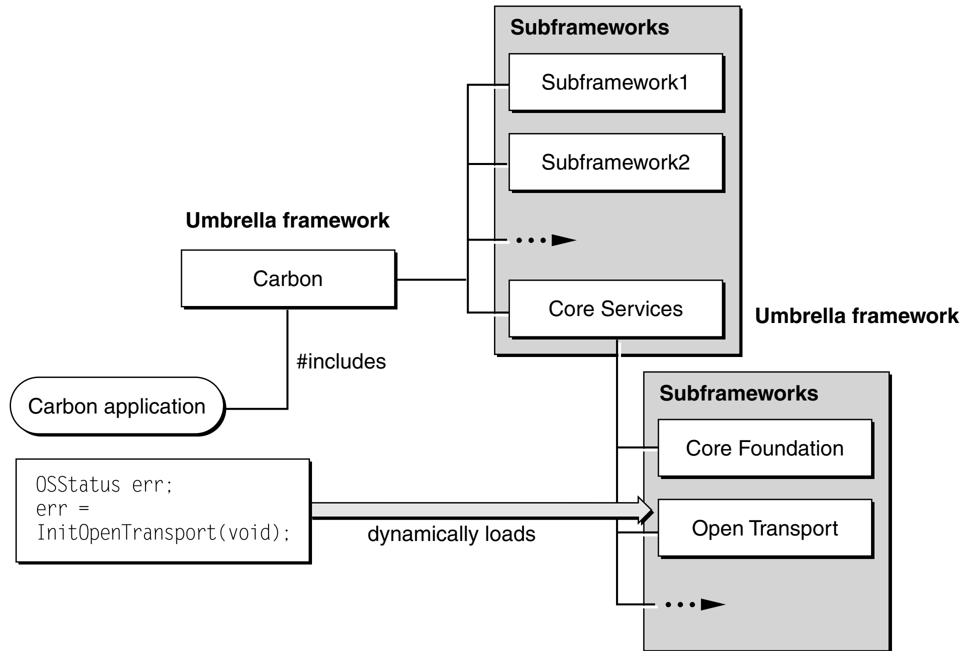
An umbrella framework simply includes and links with constituent subframeworks and other public frameworks. As the word "umbrella" implies, an umbrella framework encompasses all the technologies and APIs that define an application environment or a layer of system software. It provides a layer of abstraction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

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 118)). 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 116)).

Some umbrella frameworks include other umbrella frameworks; this is particularly this 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.

Preliminary © Apple Computer, Inc. May 2000

Figure 7-1 Relationship between framework and subframework



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.

Preliminary © Apple Computer, Inc. May 2000

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.

For umbrella frameworks Apple only supports the convention where the name of the umbrella header file matches the framework name. The `#include` (or `#import`) command is appropriately shortened to represent this convention. Thus, to include the Carbon umbrella framework, you would give the following `#include` command:

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

This convention brings with it some benefits. It permits greater portability of Carbon source code (where this single-name convention is the norm). And it results in faster compilations because the compiler (`gcc`) automatically uses the precompiled header that matches the umbrella and framework header files.

Preliminary © Apple Computer, Inc. May 2000

See "The Structure of an Umbrella Framework" (page 116) for more 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 102) in the chapter "Frameworks" 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 the manner in which an umbrella framework, 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 is

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

The convention for including umbrella header files is the following:

```
#include <Framework.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 the following contents:

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


Preliminary © Apple Computer, Inc. May 2000

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`):

```
#include <CoreFoundation.h>
#include <CoreFoundation/CFString.h>
#include <CoreFoundation/CFPlugin.h>
#include ...
```

Physically, umbrella frameworks contain their subframeworks using a structure constructed from subdirectories and symbolic links (a mechanism similar to aliases). Listing 7-1 depicts a hypothetical framework. (Symbolic links in this example are items followed by an “at” sign (@); they include the referenced path.)

Listing 7-1 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/
```


Preliminary © Apple Computer, Inc. May 2000

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 98) in the chapter "Frameworks."

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.

Preliminary © Apple Computer, Inc. May 2000

C H A P T E R 8

The Desktop and the File System

The Desktop 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. The Finder is the part of the Desktop application whose windows track and display items in the file system and mediate user access to them.

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

- application interfaces to the Desktop
- the stores of information the Desktop maintains
- how the Finder handles applications and documents
- how the Finder handles file operations that take place between volumes of different formats

In addition, this chapter discusses several important topics related to the file system, including

- the standard directory structure
- resource forks on Mac OS X
- aliases versus symbolic links

Preliminary © Apple Computer, Inc. May 2000

The Role of the Desktop

In general, the nature and role of the Desktop in Mac OS X is much the same as it is in Mac OS 9, where it is called the Finder. The Desktop 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. In Mac OS X, the Finder is that part of the Desktop that performs the file-system functions. Users launch applications and open documents through the agency of the Desktop. In a sense, it is the primary application, the one that is constantly running while users are logged into the system.

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

- 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 File Viewer, 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 the norm. Users must log into a Mac OS X system (even if they request logging in to happen automatically). Once logged in, they work in a 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. In Mac OS X must deal with the Carbon, Cocoa, Java, Classic, and (in some cases) the BSD Command 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 126) for further information.

The Desktop 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

Preliminary © Apple Computer, Inc. May 2000

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+.

Desktop Interfaces to Applications

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

Other interfaces are planned, including a suite of Apple events that applications can send to and receive from the Desktop to accomplish a number of functions, including opening documents and launching applications.

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-macos.plist`) that, when the application is built, is made part of the bundle (that is, the application package). It contains the following Finder-specific information:

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

Preliminary © Apple Computer, Inc. May 2000

This form of Finder interface is more passive than the other interfaces; all a developer must do is make this information available to the Finder. When the Finder encounters an application, it extracts the information in `Info.plist` and populates its databases with it (see “Information Stored by the Desktop” (page 122) for details).

For more information on information property lists and the keys that are specific to the Finder, see “Information Property Lists” (page 132) in the chapter “Software Configuration.” For related information, see the chapters “Bundles” (page 71) and “Application Packaging” (page 87).

Information Stored by the Desktop

The Desktop 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 Desktop 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 Desktop stores information on the file system differs from the way the Mac OS 9 stores 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.

On 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 Desktop has encountered for that user and includes information about the document types understood by each application. The Desktop extracts this information from the information property lists of applications (see “Information Property Lists” (page 121) for a summary of this information).

Preliminary © Apple Computer, Inc. May 2000

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

- The Desktop 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 using the Finder, the Desktop 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 Desktop 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 she has never attempted to open, the Desktop 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

The Finder on both Mac OS 9 and the Desktop on Mac OS X store the contents of the user's desktop in an invisible folder (named, appropriately enough, `Desktop Folder`). Although the name and invisibility of this folder are the same, the folder location and desktop semantics are quite different for the two operating systems.

- On Mac OS 9 the `Desktop Folder` is at the root of a volume; on Mac OS X the `Desktop Folder` is in the user's home directory.
- On Mac OS 9 what is displayed on the desktop is the union of each `Desktop Folder` on all volumes; on Mac OS X, what is displayed on the desktop is the contents of the `Desktop Folder` associated with the logged-in user.

Finder Attributes

Finder attributes 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 on Mac OS X recognizes fewer such attributes than the Finder on Mac OS 9. The supported attributes include

- bundle bit

Preliminary © Apple Computer, Inc. May 2000

- invisible bit
- type and creator codes
- custom icons

The attributes not supported in Mac OS X are

- icon position
- view type
- label

On 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

As described in "Information Stored by the Desktop" (page 122), the Desktop 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 and creator codes, and filename extension to identify and appropriately handle documents (including loadable bundles) and applications. The following steps outline the general logic of the Finder when it comes across an item in the file system:

1. Is it 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. Is the folder 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.

Preliminary © Apple Computer, Inc. May 2000

3. What type of bundle is it?

   The Finder obtains the type and creator codes from the information stored within the bundle (see "Anatomy of a Bundle" (page 73) in the chapter "Bundles"). 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. Is the file (including file packages from step 3) an application?

   If 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 122). 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.

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 filename extension. It uses this information as a key to look up the application (or applications) that claims the document type.

- If there is only one such application, the Finder opens the document in the application, launching it if necessary.
- If there are no applications claiming the document type, the Finder puts up a dialog box in which the user can select an appropriate application; this information is added to the application database.
- If there are multiple applications claiming the document, and the document has no associated type and creator codes (that is, the file is a non-HFS or non-HFS+ file), the Finder opens the document in one of the applications claiming that document type.

If the document has neither type and creator codes nor file extension, the Finder does nothing when the user attempts to open the document. Through the Desktop users can also select an application with which to open a document when there are multiple applications claiming that document.

Preliminary © Apple Computer, Inc. May 2000

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 reintegrating the information in the hidden file into the file's resource fork and Finder attributes. If the hidden file does not exist, the copied file has no resource fork.

Preliminary © Apple Computer, Inc. May 2000

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 127) 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 give 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`.

File-System Topics

This section covers file-system issues that are of interest to developers.

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. However, they are implemented differently, which causes them to behave differently when a referenced file or folder moves or changes.

Preliminary © Apple Computer, Inc. May 2000

Symbolic links are implemented as a reference to a path in the file system. The UFS file system tries to resolve a symbolic link by parsing the path information. Thus if you move a file that a symbolic link references to a different location in the file system, the symbolic link breaks. Therefore a symbolic link is a fragile reference to a specific file or folder.

Despite this fragility, it is useful sometimes to have a reference to a file known to always exist at a specific path in the file system, thus assigning importance to the file at that location. For these cases a symbolic link works very well; even if the file at the specified location is replaced with a new file, the symbolic link still refers to the file at that location in the file system. For example, the frameworks of Mac OS X use symbolic links extensively to implement their versioning system.

By contrast, Mac OS X implements aliases by identifying the volume and location on disk of a referenced file or folder. Each such reference has a unique identity. A a result, aliases always refer to the same file or folder regardless of where it's moved in the file system—as long as the file or folder stays on the same volume. This capability makes aliases a good way to refer to files or applications that might move around on a given volume.

However, aliases are not a good way to refer to a file or folder at a specific location in the file system. When you replace a file at a given location with a new version of the file, the alias continues to refer to the old version of the file. Also, aliases break when a referenced file is copied from one volume to another. And if the application you use to edit a referenced file writes out a new copy of the file (instead of just updating the old file), any alias to the original file is broken.

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 explictly specify the resource fork in order for this to happen.

Preliminary © Apple Computer, Inc. May 2000

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 lop off the resource fork of HFS and HFS+ files. This move eliminates the need for compressing applications to preserve resource information (using Stuffit archives, bin-hex, or similar means).

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 87)) and do 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.

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 126) makes

Preliminary © Apple Computer, Inc. May 2000

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 will be preserved.

Preliminary © Apple Computer, Inc. May 2000

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.

Preliminary © Apple Computer, Inc. May 2000

(See `http://www.w3.org/XML/1999/XML-in-10-points` for an excellent summary of XML.) Elements of the 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 131)) 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 78) of the chapter Bundles, 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 possible 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
```


Preliminary © Apple Computer, Inc. May 2000

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 information property list keys, see "Standard Keys" (page 137); for the Finder keys, see "Finder Keys" (page 139).

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 71) 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 pasteboard data (analogous to Clipboard data on Mac OS 8 and 9) 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

Preliminary © Apple Computer, Inc. May 2000

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 three roles:

- **Editor**. The application can read, manipulate, and save the type.
- **Viewer**. The application can read and present the data.
- **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

Listing 9-1 contains an example of an `Info.plist` file. This information property list, which is taken from the Sketch demonstration application, is interesting because it shows how document types for this application are specified.

Listing 9-1 The Info.plist file for the Sketch demo application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/
PropertyList.dtd">
<plist version="0.9">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleVersion</key>
    <string>1.1.0</string>
    <key>CFBundleGetInfoString</key>
    <string>Apple Sketch Application Example 1.1.0.  Copyright \U00A9 1998,
Apple Computer, Inc.</string>
    <key>CFBundleName</key>
    <string>Sketch</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.CocoaExamples.Sketch</string>
    <key>NSMainNibFile</key>
```


Preliminary © Apple Computer, Inc. May 2000

```
<string>Draw2Java.nib</string>
<key>NSHumanReadableCopyright</key>
<string>Copyright \U00A9 1998, Apple Computer, Inc.</string>
<key>CFBundlePackageType</key>
<string>APPL</string>
<key>NSJavaRoot</key>
<string>Support Files/Resources/Non-localized Resources/Java</string>
<key>CFBundleSignature</key>
<string>sktc</string>
<key>NSJavaNeeded</key>
<string>Yes</string>
<key>NSJavaPath</key>
<array>
    <string>sketch.zip</string>
</array>
<key>CFBundleShortVersionString</key>
<string>Apple Sketch Application Example 1.1.0</string>
<key>CFBundleExecutable</key>
<string>Sketch</string>
<key>CFBundleIconFile</key>
<string>Draw2App</string>
<key>CFBundleDocumentTypes</key>
<array>
    <dict>
        <key>CFBundleTypeIconFile</key>
        <string>Draw2File</string>
        <key>CFBundleTypeOSTypes</key>
        <array>
            <string>sktc</string>
        </array>
        <key>NSDocumentClass</key>
        <string>DrawDocument</string>
        <key>CFBundleTypeName</key>
        <string>Apple Sketch Graphic Format</string>
        <key>CFBundleTypeRole</key>
        <string>Editor</string>
        <key>NSExportableAs</key>
        <array>
            <string>NSPostScriptPboardType</string>
            <string>NSTIFFPboardType</string>
        </array>
```


Preliminary © Apple Computer, Inc. May 2000

```
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>sketch</string>
                <string>draw2</string>
            </array>
        </dict>
        <dict>
            <key>CFBundleTypeOSTypes</key>
            <array>
                <string>eps </string>
            </array>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>eps</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>NSPostScriptPboardType</string>
            <key>CFBundleTypeRole</key>
            <string>None</string>
        </dict>
        <dict>
            <key>CFBundleTypeOSTypes</key>
            <array>
                <string>tiff</string>
            </array>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>tiff</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>NSTIFFPboardType</string>
            <key>CFBundleTypeRole</key>
            <string>None</string>
        </dict>
    </array>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
</dict>
</plist>
```


Preliminary © Apple Computer, Inc. May 2000

The Sketch application associates with this `Info.plist` file (actually `Info-macos.plist`) an `InfoPlist.strings` file in the English-localized resource directory.

Listing 9-2 The InfoPlist.strings file for the Sketch demo application

```
{
    CFBundleName = "Sketch";
    CFBundleShortVersionString = "Apple Sketch Application Example 1.1.0";
    CFBundleGetInfoString = "Apple Sketch Application Example 1.1.0.
Copyright \U00A9 1998, Apple Computer, Inc.";
    NSHumanReadableCopyright = "Copyright \U00A9 1998, Apple Computer, Inc.";

    "Apple Sketch Graphic Format" = "Apple Sketch Graphic Format";
    "NSPostScriptPboardType" = "NSPostScriptPboardType";
    "NSTIFFPboardType" = "NSTIFFPboardType";
}
```

Standard Keys

Mac OS X defines a small set of standard keys. Some of these keys are given default values by the integrated development environment.

CFBundleInfoDictionaryVersion. Used to support future versioning of the `Info.plist` format. It is automatically generated by the development environment when you are building a bundle.

CFBundleExecutable. The name of the main executable for the bundle. 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 (in the case of a framework, 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.

CFBundleIdentifier. The unique identifier string for the bundle. This identifier should be in the form of a Java-style package name, for example `com.apple.foo.bar`. The bundle identifier can be used to locate the bundle at runtime. The preferences system uses this string to identify applications uniquely.

Preliminary © Apple Computer, Inc. May 2000

CFBundleVersion. A Mac OS `'vers'` resource style version number. The value of this key should be a string. The value is application-specific; however, if the standard form "2.5.3d5" is used, the system's bundle routines can correctly retrieve the value. If the value is an arbitrary number, the bundle routines treat it as a string, but then the routines are not guaranteed to return the proper numeric representation.

CFBundleDevelopmentRegion. The "native" region for the bundle. Usually this is the native language of the person who wrote the bundle. The development region is used as the last resort if a resource cannot be located for the user's preferred region or language.

The following keys are applicable to Cocoa bundles only:

NSJavaNeeded. If given a "true" boolean value, then the Java VM is loaded and started up, if necessary. A "true" boolean value can be either a CFBoolean object set to true (in XML) or a "YES" string value.

NSJavaPath. An array of paths to classes whose components will be preceded by `NSJavaRoot` if they are not absolute locations. The development environment (or, specifically, its makefiles) automatically maintains the values in the array.

NSJavaRoot. A string specifying the root of the directory location where the application's Java classes are.

NSMainNibFile. The name of an application's main nib file. A nib file is an Interface Builder archive containing the description of a human interface along with connections between objects of that interface. The main nib file is automatically loaded when an application is launched. By default, this filename is the name of the application with an extension of `.nib`.

NSPrincipalClass. The name of a bundle's principal class. The principal class is designated the main class because of its central relation to other classes in the bundle. By default, this name is the application name.

NSServices. An array of dictionaries specifying the services provided by an application. Keys for this subdictionary are NSPortName, NSSendTypes, NSMenuItem, and NSMessage.

NSHumanReadableCopyright. A string containing copyright information to put in the Show Inspector dialog box. This key is usually in the `InfoPlist.strings` file because it needs to be localized.

Preliminary © Apple Computer, Inc. May 2000

NSHelpFile. The name of the bundle's HTML help file. This file is located in the bundle's localized resource folders or in the `Nonlocalized Resources` directory.

Finder Keys

These keys are used by the Mac OS X Finder to store important information about a bundle. Among other things, the Finder uses these properties to locate and display an application's icon and recognize associated document types.

CFBundleName. The short name of the bundle suitable for displaying in various places in the user interface, such as the menu and the About box. This key is usually in the `InfoPlist.strings` file because it needs to be localized.

CFBundlePackageType. The four-letter type code for the bundle. This is `'APPL'` for applications, `'FMWK'` for frameworks, and `'BNDL'` for generic bundles. You can choose a more specific type code for generic bundles.

CFBundleSignature. The four-letter creator code for the bundle.

CFBundleIconFile. The filename of the bundle resource that contains the icon to be used to display this bundle in the Finder (or other applications). The 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" on Macintosh).

CFBundleShortVersionString. A human-readable description of the bundle's version. This should be more than just the string that can be generated from the `CFBundleVersion` key, if present. This key is usually in the `InfoPlist.strings` file because it needs to be localized.

CFBundleGetInfoString. A human-readable plain text string displayed in the Show Inspector dialog box. This key is usually in the `InfoPlist.strings` file because it needs to be localized.

CFBundleGetInfoHTML. A human-readable HTML string displayed in the Show Inspector dialog box. This key is usually in the `InfoPlist.strings` file because it needs to be localized. You can specify this key-value pair instead of the plain text `CFBundleGetInfoString` if you want a richer representation. If `CFBundleGetInfoString` and `CFBundleGetInfoHTML` are both present, `CFBundleGetInfoHTML` is used.

Preliminary © Apple Computer, Inc. May 2000

CFBundleDocumentTypes. An array of the type definitions for any document types an application understands. Each type definition is a dictionary in the array. These keys are supported in the type-definition dictionary:

- **CFBundleTypeName**. The abstract name for the document type, which must be present for the type to be valid. This is the main way to refer to a type, and is used by the system when data is on the pasteboard. To ensure uniqueness, it is recommended that you use a Java-package style identifier. This identifier is also used as a key in the `InfoPlist.strings` file to provide the human-readable version of the type name. If the type is a system type, you can use one of the symbol names for common pasteboard types:

  NSStringPboardType
  NSFilenamesPboardType
  NSPostScriptPboardType
  NSTIFFPboardType
  NSRTFPboardType
  NSTabularTextPboardType
  NSFontPboardType
  NSRulerPboardType
  NSFileContentsPboardType
  NSColorPboardType
  NSPICTPboardType
  NSPDFPboardType
  NSURLPboardType

- **CFBundleTypeIconFile**. Specifies the filename (minus the extension) of the resource in the bundle that contains the icon the Finder should display for the type. The 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” on Macintosh).
- **CFBundleTypeRole**. Defines the application’s role with respect to the type. The role can be `Editor`, `Viewer` or `None`. An editor can read, manipulate, and write the type, a viewer can only read the type, and an application that wants to simply declare a type without claiming to be able to read or write it can use `None`.
- **CFBundleTypeOSTypes**. An array of four-letter type codes that map to this type.

Preliminary © Apple Computer, Inc. May 2000

- **CFBundleTypeExtensions**. An array of filename extensions that map to this type.
- **NSDocumentClass**. The NSDocument subclass used to instantiate instances of this document. Used for Cocoa applications only.
- **NSExportableAs**. An array of other types that documents of this type can be exported as (write-only types). Used for Cocoa applications only.

CFBundleURLTypes. An array of dictionaries similar to `CFBundleDocumentTypes`, but it describes URL schemes that the application can handle. These keys are supported in a URL-type dictionary:

- **CFBundleURLName**. 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.
- **CFBundleURLIconFile**. Specify the filename (minus the extension) of the resource in the bundle that contains the icon to be used for this type.
- **CFBundleURLSchemes**. An array of URL schemes handled by this type (`http`, `ftp`, and so forth)

The Preferences System

Preferences are application or system options that users can select to customize their working environment. For example, automatic save, default font, and smart quotes are common preferences for document-based applications. Almost all applications need to store and retrieve preferences. The preferences system of Mac OS X not only let users customize the behavior of applications and system software, but it provides a way to preserve preference settings across multiple launches. Preferences are not limited to applications; frameworks and libraries can write and read preferences including, on occasion, user preferences. For creating, writing, reading, and removing preferences, use Core Foundation’s Preference Services or, for Cocoa developers, the NSUserDefaults class.

Preliminary © Apple Computer, Inc. May 2000

Important
You should not store data needed to configure an application at launch time as a preference. The assumption with user preferences is that they are not critical; if somehow they are lost, the application can recreate the default set of preferences. Initial configuration information *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

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`.

The files in `Library/Preferences` take 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 Keys" (page 137) 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

Preliminary © Apple Computer, Inc. May 2000

are com.apple.Finder, com.adobe.Photoshop, 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 wish, you can directly modify these XML property lists to add or change application preferences. By doing so, however, you can introduce editing errors into the XML; if this happens, the application might not be able to load the file and so it will lose all its preferences. If you must edit preferences files, use the Property List Editor application.

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 9-1 shows all of the preference domains, listed in the order they are searched when the preference system attempts to locate a preference value.

Table 9-1 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

Preliminary © Apple Computer, Inc. May 2000

Table 9-1 Preference domains in search order (continued)

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 will be 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 fuller description, 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.

Preliminary © Apple Computer, Inc. May 2000

C H A P T E R 1 0

Inter-Environment Issues

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) on Mac OS X is to manage memory, address

Preliminary © Apple Computer, Inc. May 2000

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, for more on Multiprocessing Services tasks.

Threading Packages

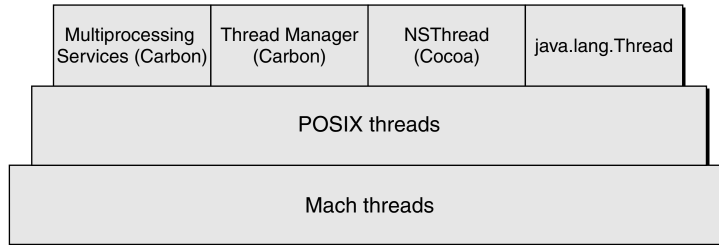
A thread is an execution context within a process (see "Tasks and Processes" (page 145)). It is associated with a call stack and a processor's state. A thread shares virtual address space and other task-wide resources with other threads of the process. Threads are scheduled to run preemptively or, with symmetric multiprocessing, concurrently. User threading models 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.

Preliminary © Apple Computer, Inc. May 2000

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 10-1 depicts these packages and the dependencies.

Figure 10-1 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.

Preliminary © Apple Computer, Inc. May 2000

Threading package	Description
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 10-1 (page 147) 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 Carbon's Multiprocessing Services and Cocoa's NSThread class 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 only be cooperatively scheduled.

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).

Preliminary © Apple Computer, Inc. May 2000

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.

Except for 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

In Mac OS X, a program has a number of ways to communicate with other programs. It can let them know what it is doing, request data or a service from them, or send whatever information might be interesting to them. These mechanisms for interprocess communication each have their own purposes, limitations, and intended scenarios. Some are more suitable than others for code written at a certain level of the system; for example, kernel extensions would not make use of Apple events.

This section summarizes each of the mechanisms or APIs for interprocess communication in Mac OS X and offers some guidelines for their use.

- **Apple events and AppleScript**. As described in "Apple Events" (page 66), an Apple event is a high-level event that a process such as an application can send to another process or even to itself.

  To use Apple events, the receiving (server) process and, generally, the sending (client) process must contain code specific to these events. That is, they must define and implement handlers for all possible Apple events that they might expect to receive. An Apple-event handler extracts data from an Apple event, performs the requested command, and (usually) returns a result. A centralized Apple event registry defines the standard suites of Apple events: Required, Core, and functional-area suites such as Text and Database. An application can define its own custom Apple events, but these should be recorded in the registry.

Preliminary © Apple Computer, Inc. May 2000

- **Distributed notifications**. Distributed notifications are messages that any process can post to a per-machine "notification center," which in turn forwards them to any observers (processes) on that same machine that are interested in receiving the notification. Included with the notification is an identifier of the sender and, optionally, a dictionary containing additional information. The distributed-notification mechanism is implemented by Core Foundation Notification Services (see "Core Foundation" (page 64)) and by Cocoa's NSDistributedNotificationCenter class.

  Distributed notifications have several features that make them a good choice for communicating information between processes: multicasting-type behavior, asynchronicity, and coalescing and suspension of notifications. They are ideal for ensuring that applications and other processes on a computer are aware of each other's behavior so they can respond appropriately. Unlike Apple events, however, you cannot use distributed notifications to send messages to a remote process (that is, a process on a different machine). In addition, Notification Services provides no way to respond directly to the sender of a notification. A process can use distributed notifications to post notifications to itself, although this is an expensive operation.

- **BSD mechanisms**.
  - Signals. Signals are preset conditions that can be delivered to processes. A process examines the condition each time its time slice comes around and act accordingly, often by delivering the process to a specified handler.
  - Shared memory and memory-mapped files (both POSIX semaphores for synchronization).
  - Sockets and pipes (both unnamed as well as named pipes).

  The primary users of these BSD facilities are typically operating-system services that need compatibility with other BSD-based operating systems. Note that the behavior of these facilities might differ between BSD systems. Consult the man pages or other BSD documentation for further information.

- **Mach messaging**. Processes can send messages to other processes using the Mach messaging infrastructure. Developers of all types of software are discouraged from using Mach messaging directly if other alternatives are available.

Preliminary © Apple Computer, Inc. May 2000

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 102) in the chapter "Frameworks" for a description of the dynamic link editor.

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.

Preliminary © Apple Computer, Inc. May 2000

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. It 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, nonconstant 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 runtimes 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 to a base address. This method of access is known as GOT.

Preliminary © Apple Computer, Inc. May 2000

Vector Libraries

All system frameworks on 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 CFPlugIn, 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 dyld.

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 plug-in. You can create a dyld-based plug-in 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 plug-in APIs (specifically Core Foundation Plug-in Services) to load the plug-in and use it to call into the non-Carbon framework.

Preliminary © Apple Computer, Inc. May 2000

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 87)) 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.

Preliminary © Apple Computer, Inc. May 2000

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**.

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**.

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.

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**.

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.

Preliminary © Apple Computer, Inc. May 2000

Carbon An application environment on 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 on 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 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 have the same types of instance variables and have access to the same methods belong to the same class.

Classic An application environment on 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.

Cocoa An advanced object-oriented development platform on 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.

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.

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**.

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.

DVD Digital Versatile Disc or Digital Video Disc. 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 Dynamic link editor. The library manager for processes based on Mach-O object files.

Preliminary © Apple Computer, Inc. May 2000

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.

Ethernet A high-speed local area network technology.

exception An interruption to the normal flow of program control caused by the program itself.

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

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 spread of unauthorized data).

fork A stream of data that can be opened and accessed individually under a common filename. The Mac 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. In BSD, `fork` is a system call that creates a new process.

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 file names 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.

Preliminary © Apple Computer, Inc. May 2000

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**.

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.

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**.

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

Mach-O Mach object file format.

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 thread APIs 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 make to build an executable version of an application. A makefile details the files, dependencies, and rules by which the application is built.

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.

memory-mapped files A facility that maps virtual memory onto a physical file. Thereafter, any access to that part of virtual memory causes the corresponding page of the physical file to be accessed. The contents of the file can be changed by changing the contents in the memory.

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

Preliminary © Apple Computer, Inc. May 2000

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 computer 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.

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.

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.

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

nonretained window A window without an off-screen buffer for screen pixel values.

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.

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.

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.

PEF Preferred Executable Format. An executable format understood by the Code Fragment Manager.

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

Preliminary © Apple Computer, Inc. May 2000

port In Mach, a secure unidirectional channel for communication between tasks running on a single system. 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.

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**.

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

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 POSIX Threads package (BSD).

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

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.

retained window A window with an off-screen 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 three roles: `Editor` (reads and modifies), `Viewer` (can only read), 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.

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

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

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.

Preliminary © Apple Computer, Inc. May 2000

socket In BSD-derived systems, a socket refers to different entities in user and kernel operation. 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. In AppleTalk protocols, a socket serves the same purpose as a "port" in IP transport protocols.

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 framework in which threads run. See also **thread**.

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.

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.

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.

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**.

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. Specifically, its disk layout is not compatible with other BSD UFS implementations.

UDF Universal Disk Format. The file-system format used in DVD disks.

Preliminary © Apple Computer, Inc. May 2000

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.

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**.

virtual address An 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**.

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 memory (VM) The use of a disk partition or a file on disk to provide the same facilities usually provided by RAM. The virtual-memory manger on Mac OS X provides 32-bit (minimum) protected address space for each task and facilitates efficient sharing of that address space.

Preliminary © Apple Computer, Inc. May 2000

Index

A

address space 146
ADSP 36
advanced virtual memory 25
AFP 28
alias
  definition 127
anti-aliasing 30
antialiasing 54
Apple event 66, 149
Apple events 23
Apple Help 94
Apple Type Solution 33
AppleScript 23, 126, 149
AppleTalk 36, 67
application 78, 87
  and plug-ins 93
  and private frameworks 90
  and resource fork 128
  and shared frameworks 91
  exported services 23
  help 94
  helper 92
  main bundle 79
  packaging 20, 87
  preferences 95
application database 122
application environment 19, 120
application environments 39
application extensibility 22
Application Kit 47, 48, 49
application package 21
Application Services 39, 42
applications
  and the Finder 124
Aqua 15, 120
  user interface 17
architecture 39, 41
  Java environment 50
assistant 92
ATP. *See* Apple Type Solution
ATP/ASP 36
ATSUI 33
AWT 49, 50

B

Base Services 65
bit depth 30
BOOTP 35
BSD 26, 44
  interprocess communication 150
BSD command 41
bundle 21
  and property list 132
  and resources 73, 76
  application 87
  benefits 72, 88
  definition 71
  information property list 132
  loadable 80
  search algorithm 82
  structure 73
  types 71, 78
  versioned 73, 78
bundle bit 77
Bundle Services 65
bytecode compiler 49

Preliminary © Apple Computer, Inc. May 2000

C

Carbon 15, 20, 42, 45
  and resources 46
  core managers 62
  documentation 47
  event handling 69
  general changes from Mac OS 9 46
  hardware interfaces 46
  memory management 45
  new managers 46
  replacement managers 46
Carbon Event Manager 61, 69
Carbon Process Manager 146
central directory 91
chip architectures 72
Classic 15, 41, 120
Cocoa 15, 20, 42, 47, 49
  event handling 69
Collection Services 65
column view 19
compiler
  JIT bytecode 49
compositing 54
configuring software 131
converter 59
cooperative multitasking 25
cooperative threading 25
coordinate system 54
copy operation 126
Core Foundation 64
Core Graphics Rendering 30, 51, 54
Core Graphics Services 30, 51, 53
Core Services 39, 43, 51, 53, 62

D

Darwin 15, 24
data fork 128
Desktop application 119, 120
Desktop Folder 123
device driver 44
  I/O Kit 26
DHCP 35
display independence 54
distributed notifications 150
DNS 35
Dock 18
document
  abstract type 133
  and resource fork 129
  concrete type 133
  resources 96
  role 134
document configuration 133
documents
  and the Finder 125
DVD 28
dynamic linking 80

E

Ethernet 34
event
  handling 61, 69
  low-level 68
event port 69
event queue 68
EventRefs 69
executable
  and bundles 72

F

file operations 126
file package 77, 88
file systems 27, 44
Finder 19, 119, 120
  and aliases 127
  and applications 89
  and bundles 77
  and resource fork 126
  and symbolic links 127
  application database 122

Preliminary © Apple Computer, Inc. May 2000

Finder (*continued*)
bundle configuration 139
copying files 126
handling applications 124
handling documents 125
Finder attributes 123
FireWire 38
Foundation 47, 48, 49
frame buffer 30
framework 79, 80, 89
and applications 89
definition 112
in bundles 77
including 115
linking against 115
private 77, 90, 112
public 112
shared 91
subframework 113
types of 112
umbrella 111, 113

G

graphics and windowing environment 51

H

hardware 37
hardware interfaces
and Carbon 46
Help 72
help files 94
Help Viewer 94
helper application 92
HFS 27, 121
HFS+ 27, 121

I

I/O Kit 53, 68
I/O module 59
IDE 37
IEEE 1394 38
Info.plist . *See* information property list
InfoPlist.strings 133
information property list 74, 122, 131, 132
and Finder 121
Finder keys 139
standard keys 137
installation 24
internationalization 21, 72, 81
Internet 24
interprocess communication 149
IP 67
IP aliasing 36
IP routing 36
IPC 68
ISO 3166 81
ISO 639 81
ISO 9660 28, 44

J

Java 42, 47, 49
application framework 50
architecture 50
basic packages 50
event handling 69
runtime 49
tools 49
java.lang.Thread 148
JDirect 49
JNI 49
Job Manager 59
job ticket 59

Preliminary © Apple Computer, Inc. May 2000

K

kernel environment 40, 43, 147
keyboard focus 61

L

layered compositing 54
layers of system software 41
LDAP 35
loadable bundle 79, 80, 93
localization 21, 88
localized resource 21
localized resources 81
localized strings 82

M

Mac OS Extended 44, 121
Mac OS Extended Format 27
Mac OS Standard 44, 121
Mac OS Standard Format 27
Mach 24, 43
  messaging 150
  thread 147
memory
  and Carbon 45
  protection 43
minimum resolution 30
move operation 126
multihoming 36
multiple users 20, 120
Multiprocessing Services 146, 147

N

NetInfo 21
network 27
Network Kernel Extensions 27, 44
network protocol stack 34
networking 44
NFS 28, 44
NKE. *See* Network Kernel Extensions
notification
  file system 28
Notification Services
  distributed notifications 66
NSEvent 69
NSTask 146
NSThread 148
NTP 35

O

Objective-C 47
opaque types 64
Open Source 29
Open Transport 36, 67
OpenGL 15, 31, 52

P

palette 79
PAP 35
pasteboard 62
PBM
  See printer browser module 59
PDE
  See printing dialog extensions 58
PDF 29, 31, 55
Personal Web Sharing 37
PJC. *See* Print Job Creator
plug-in 22, 79, 93
Plug-in Services 65
Portable Document Format. *See* PDF
POSIX threads 147
PostScript 29, 55
  printers 33
PPP 35
preemptive multitasking 25, 43
preemptive thread 146

Preliminary © Apple Computer, Inc. May 2000

preemptive threading 25
Preference Services 66
Preferences 131
preferences 95, 141
Print Center 33
Print Job Creator 58
print preview 33
print spooling 33
PrintCenter 56, 58
printer browser module 59
printer discovery 59
printer module 59
printing 55, 56
  architecture 57
  converter 59
  data flow 60
  I/O module 59
  Job Manager 59
  job ticket 59
  Print Job Creator 58
  PrintCenter 58
  printer browser module 59
  printer discovery 59
  printer module 59
  printing dialog extensions 58
  Queue Manager 59
  user interface 56
printing dialog extensions 58
printing system 32
private framework 112
PrivateFrameworks folder 112
process 61, 145
  BSD 146
Process Manager 61
property list 66, 131
Property List Services 66
protected memory 25
pthreads. *See* POSIX threads
public framework 112

Q

Quartz 15, 29, 51, 52
Queue Manager 59
QuickDraw 30, 31, 38, 55
QuickTime 15, 32, 38, 42, 49, 52
QuickTime for Java 49
quotas 28

R

raster printers 33
real-time support 25
resource fork
  and Finder 126
  and Mac OS X 128
resources
  in bundles 73, 76
  localizing 81
RTP 32
RTSP 32
run loop 69
Run Loop Services 66

S

scripting 23
SCSI 37
searching for resources 82
serial 34
Services menu 23
shared library 89
Sherlock 72
Sherlock 2 24
Single Window Mode 17
SLP 35
software configuration 131
String Services 65
strings
  localized 82

Preliminary © Apple Computer, Inc. May 2000

subframework 113
  linking restrictions 118
Swing 49, 50
symbolic link
  definition 127
symmetric multiprocessing 43, 146
System framework 42, 147
system software
  layers 41

T

task 145
  Mach 145
TCP/IP 35, 67
thread 145
  definition 146
  layering of packages 148
  scheduling 146
Thread Manager 148
Thread Packages 146
tool 92
translucency 54

U

UDF 28
UDP/IP 35
UFS 28, 44
umbrella framework 45, 111, 113
  including 115
  linking against 115
  memory footprint 116
  purpose 113
  structure 116
URL Services 65
USB 37
Utility Services 66

V

vector information 54
Velocity Engine 30, 38
VFS . *See* Virtual File System
Virtual File System 27, 43, 44
virtual machine 49
virtual memory 43
VM
  See virtual machine 49
volume format 120

W

window
  active 61
window composition 54
window server 53
  event queue 68

X, Y, Z

XML 131
XML Parser 66

Preliminary © Apple Computer, Inc. May 2000