AFFIDAVIT OF NATHANIEL E FRANK-WHITE

1. I am a Records Request Processor at the Internet Archive. I make this declaration of my own personal knowledge.

2. The Internet Archive is a website that provides access to a digital library of Internet sites and other cultural artifacts in digital form. Like a paper library, we provide free access to researchers, historians, scholars, and the general public. The Internet Archive has partnered with and receives support from various institutions, including the Library of Congress.

3. The Internet Archive has created a service known as the Wayback Machine. The Wayback Machine makes it possible to browse more than 450 billion pages stored in the Internet Archive's web archive. Visitors to the Wayback Machine can search archives by URL (i.e., a website address). If archived records for a URL are available, the visitor will be presented with a display of available dates. The visitor may select one of those dates, and begin browsing an archived version of the Web. Links on archived files in the Wayback Machine point to other archived files (whether HTML pages or other file types), if any are found for the URL indicated by a given link. For instance, the Wayback Machine is designed such that when a visitor clicks on a hyperlink on an archived page that points to another URL, the visitor will be served the archived file found for the hyperlink's URL with the closest available date to the initial file containing the hyperlink.

4. The archived data made viewable and browsable by the Wayback Machine is obtained by use of web archiving software that automatically stores copies of files available via the Internet, each file preserved as it existed at a particular point in time.

5. The Internet Archive assigns a URL on its site to the archived files in the format http://web.archive.org/web/[Year in yyyy][Month in mm][Day in dd][Time code in hh:mm:ss]/[Archived URL] aka an "extended URL". Thus, the extended URL http://web.archive.org/web/19970126045828/http://www.archive.org/ would be the URL for the record of the Internet Archive home page HTML file (http://www.archive.org/) archived on January 26, 1997 at 4:58 a.m. and 28 seconds (1997/01/26 at 04:58:28). The date indicated by an extended URL applies to a preserved instance of a file for a given URL, but not necessarily to any other files linked therein. Thus, in the case of a page constituted by a primary HTML file and other separate files (e.g., files with images, audio, multimedia, design elements, or other embedded content) linked within that primary HTML file, the primary HTML file and the other files will each have their own respective extended URLs and may not have been archived on the same dates.

6. Attached hereto as Exhibit A are true and accurate copies of screenshots of the Internet Archive's records of the archived files for the URLs and the dates specified in the attached coversheet of each printout.

7. Attached hereto as Exhibit B are true and accurate copies of the Internet Archive's records of the archived files for the URLs and the dates specified in the attached coversheet of each file.

8. I declare under penalty of perjury that the foregoing is true and correct.

DATE: 12/12/2022

Please see attached All Purpose Jurat form for additional Notary Events

Nathaniel Frank-White

Nathaniel E Frank-White

JURAT ATTACHMENT

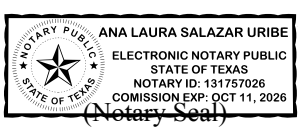
A notary public or other officer completing this certificate verifies only the identity of the individual who signed the document to which this certificate is attached, and not the truthfulness, accuracy, or validity of that document.

STATE OF Texas }

COUNTY OF Harris }

The foregoing instrument was subscribed and sworn before me this date of 12/12/2022, by Nathaniel Frank-White

This notarial act was an online notarization.



(Notary Seal)

Notary's Signature Ana Laura Salazar Uribe

Registration No.: 131757026

Commission Expiration Date: October 11, 2026

EXHIBIT A

https://web.archive.org/web/20100328220626/http://www.panoramio.com:80/help/adding_photos

Panoramio Help

Google™ Custom Search Search
Search Panoramio help articles.

Help topics

- Help home
- Viewing photos
- **Adding photos to Panoramio**
  - Adding a single photo
  - Adding multiple photos
  - Mapping photos
  - Uploading geocoded photos
  - Viewing your uploaded photos
- Tagging photos
- Understanding popularity in Panoramio
- Getting your photos into Google Earth
- Changing previously posted photos
- Showing your photos to friends
- Sharing and community
- Embedding a Panoramio map into your web page
- Usernames, passwords, and accounts
- Getting more involved with Panoramio
- Becoming a Panoramio translator
- Troubleshooting
- Panoramio policies and legalities
- Spam in comments
- Suggesting a new cool place

Adding photos to Panoramio

Panoramio's limitations are minimal, designed to let you upload lots of photos with little effort! This table shows how many photos you can upload and what size they can be.

Limitation	Maximum	Comments
Number of photos	The number of photos isn't limited, but their total disk space can't exceed 2 Gbytes.	Approximate equivalent number of photos: • 4 megapixel camera: 1000 photos • 7 megapixel camera: 500 photos
Size of photos	5 MBytes per photo	Please resize a photo if it's larger than 5 MBytes. For all other photos, the Panoramio server automatically resizes the photos for you.

Adding a single photo

To upload a photo, you'll first need to be registered. If you haven't registered yet, now is the time to sign up for Panoramio.

To upload a photo:

1. Click **Upload your photos** if you're looking at the Panoramio home page, or click the **Upload** tab from most other pages.
2. Browse to select the photo from your computer.
3. Give the photo a title, if you want.
4. To map your photo now, click **Map this photo**. For more information on mapping photos, see Mapping photos. You can map your photo later, or start now and cancel if you change your mind.
5. Click **Finish**.

Adding multiple photos

To add multiple photos:

1. Upload the first photo as described above.
2. While the first photo is uploading, browse for another photo and select it. Keep browsing and selecting photos until you've selected as many photos as you'd like.
3. Click **Finish** when you've selected all the photos you want to upload. There's no need to wait for the uploads to finish.

You can map the photos as you upload them, or map them later. When you map multiple photos, notice that the default location of a photo is the same as the location of the previous photo, so it's easy to map multiple photos taken during one photo shoot.

To significantly change the map location from the location of the previous photo, look for the **Location** indicator above the map and click **[change it]**. This is an example of what to look for:

> **Location:** Zurich, Switzerland **[change it]**

Panoramio doesn't support batch uploads. Each photo must be individually uploaded to ensure the best quality and geolocation accuracy.

Mapping photos

To map a photo, you associate it with a location. You can identify the location by searching for the location on the map and manipulating the map image, or by entering the coordinates.

New Panoramio users often ask whether to associate a photo with the location in which they were standing or the location that they were viewing. Panoramio prefers that you map photos to the location where you were standing when you took them. As an example, if you took a photo of a city while standing on a hill above the city, you associate the photo with the hill, although it shows the city.

After you've mapped the photo, others can go to the actual location and see the same view. You might find that if you don't map the photo precisely, other members will suggest corrections to your locations.

If you are an advanced user who geocodes photos before uploading them, skip this section and refer to Uploading geocoded photos.

How to map a photo

To map a photo:

1. Do one of the following:
   - If you are uploading a new photo, while it's being uploaded, or after it's done, click **Map this photo**.
   - If you are mapping a photo that's already in Panoramio, go to your personal page (**Your photos**), and click to view the photo.
2. Specify the location of the photo, using the instructions in How to specify map locations. If the location you specify is ambiguous, Panoramio displays some options.
3. Click one of the options. Panoramio then displays a map with the specified location marked.
4. Zoom in and move the marker if necessary to correct the location or make it more exact.
5. Do one of the following:
   - If this is a new photo, click **Finish.**
   - If this photo was already in Panoramio, click **Save position.**

How to specify map locations

You can specify a map location by searching for it, or by entering coordinates.

To search, note these options:

- You can enter a standard place name, such as city and country, or city, state or province, and country.
- You can enter the name of a famous street, such as Third Avenue, New York or Ramblas, Barcelona.
- You can enter a well-known place name, such as Eiffel Tower, Mount Fuji or Cape of Good Hope.
- You can enter the name in your preferred language. For example, you can enter Saragossa or Zaragoza; Nueva York or New York; Köln or Cologne; and Leipzig or Lipsia.
- You should add further identification if a place has a common name. For example, add the name of a province, state, or country to distinguish the place name. For example, you might enter Cox, Spain; Cox, California; or Cox, England.

To provide coordinates:

- Supply latitude and longitude information in either decimal or degree-minute-second (DMS) notation.
- Use positive coordinates to indicate Northern latitudes and Eastern longitudes. Use negative coordinates to indicate Southern latitude and Western longitude. Do not prefix longitudes and latitudes with N, S, E, or W.

Here's an example of the ways you can enter coordinates.

- Decimal example: 40.56345, -3.45678
- DMS example: 37° 4' 39.11", -110° 57' 53.09"

Uploading geocoded photos

When a digital camera saves a photo, it also saves some additional information, such as camera settings, date, shutter speed, or scene information. This data is saved in exchangeable image file (EXIF) format.

Some cameras have GPS receivers or can accept external GPS connectors, so that they can store the geolocation information with a photo. You can also process photos before uploading them to Panoramio to add geolocation information to the EXIF data, using hardware or software solutions.

If you have geocoded a photo before uploading it, you don't have to map it in Panoramio: the information contained with the photo lets Panoramio automatically map the photo. If you manually map a photo that's already geocoded, the manual mapping supersedes the geolocation information and the original geolocation information can't be retrieved.

Some EXIF data is not readable by Panoramio. Try using a different EXIF editor, such as Exifer.

Viewing your uploaded photos

As soon as you've uploaded your photos to Panoramio you can view them on Panoramio's map, and in a local Google Earth KML file, where your photos appear as miniatures.

To view your photos on Panoramio's map:

- While viewing Panoramio's map, click the **Recent** tab or the **Your photos** tab.

To view your photos as they will appear in Google Earth:

- Refer to How to preview your photo in Google Earth.

https://web.archive.org/web/20101126084240/http://www.panoramio.com:80/map

Panoramio

You must be logged to access this page

Popular Recent

Also show photos not selected for Google Earth

Popular photos in Google Earth

https://web.archive.org/web/20101126084240/http://www.panoramio.com/map#lt=45.460131&ln=-84.726563&z=13&k=0&a=1&tab=1

You must be logged to access this page

Popular Recent

Also show photos not selected for Google Earth

Popular photos in Google Earth

https://web.archive.org/web/20100212221924/http://www.apple.com/aperture

Taking photos. Further.

The new Aperture 3 gives you powerful yet easy-to-use tools to refine images, showcase your photography, and manage massive libraries on your Mac. It's pro performance with iPhoto simplicity.
Learn more ▸

What is Aperture?

It's the way to better photos on a Mac. With tools to retouch photos, organize libraries, share work online, and print professionally designed books. Learn more

New in Aperture 3

Organization with Faces and Places. Brushes and adjustment presets to perfect and enhance images. True full-screen browsing. And over 200 more new features. Learn more

Aperture in Action

With Aperture 3, photojournalist Bill Frakes creates a multimedia slideshow that combines still photos, audio, and HD video to document people's everyday lives. Watch video

Aperture in Action

Aperture 3 helps *National Geographic* photographer Jim Richardson connect the places he's been with the pictures he's taken, using GPS mapping. Watch video

Aperture in Action

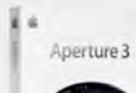
For his documentary project Wisdom of New York, professional photographer Doug Menuez uses Aperture 3 to manage his photos on location and sync edits back in the studio. Watch video

Go from iPhoto to Aperture

You've taken some amazing shots and iPhoto has been a solid photo assistant. Now you're ready to go further. The move to Aperture 3 is designed to be seamless. Learn more

200+
New Features

More power, more ease of use, and more possibilities. Learn More ▸



Buy Aperture 3 now.
Find an Apple Retail Store.
Find your local authorized reseller.
Get Apple education pricing.

Download the 30-day free trial.
Download Free Trial

Call 1-800-MY-APPLE.
View system requirements.

https://web.archive.org/web/20100211033049/http://www.macworld.com/article/146231/2010/02/aperture3.html

Digital Photo

Feb 9, 2010 5:40 am | 0 Comments | 0 Recommendations | ShareThis

Apple releases Aperture 3

Updated interface, Faces, Places highlight new features

by Jason Snell, Macworld.com

Apple may not be at Macworld 2010 this week, but the company still managed to announce a new product just in time for the trade show. On Tuesday Apple announced Aperture 3, the latest version of the company's $199 pro-level photo-editing and management software.

"We've added over 200 new features, so it's a pretty big release," Kirk Paulsen, senior director of photo applications at Apple, told *Macworld*. "And we've addressed the entire workflow, from import all the way to output. Our focus has been on keeping the pro performance, but streamlining and simplifying Aperture for iPhoto users who want to do more. So the yin-and-yang of performance and simplicity is what we focused on."

PEOPLE WHO READ THIS ALSO READ:

Recent Digital Photo Posts

Feb 9
Canon adds fun features to PowerShot

Feb 9
Apple releases Aperture 3

Feb 8
Canon announces EOS Rebel T2i

Digital Photo home

View all Macworld blogs

The update ($99 for existing Aperture users; a free trial will also be available) includes advanced versions of the Faces and Places features previously seen in iPhoto '09. There's a new image-refinement tool called Brushes, which allows users to paint adjustments directly on to images. And a new Projects feature adds flexibility. Aperture 3 will run in 64-bit mode on Snow Leopard and 64-bit processors.

Paulsen said the Aperture 3 interface "looks more iPhoto-like," with larger and easier-to-read type and streamlined buttons and sliders. "Everything's bigger and redesigned," he said. And iPhoto '09 users can import their libraries directly into Aperture 3, with all existing Events, Faces, and Places preserved.

Faces and Places

When iPhoto '09 was released, Paulsen said, "the Aperture discussion boards lit up, asking when they could see" Faces and Places within Aperture. With Aperture 3, those features have been added—but they've also been "extended and enhanced for prosumer photographers—and pros, for that matter," according to Paulsen.

Under the hood, Aperture 3 uses the same face-recognition technology as iPhoto '09. But Paulsen said that in Aperture the user has more control, such as being able to constrain face-scanning to individual projects—for example, a wedding photographer can limit face scanning to a single

Macworld's Product Guides


Mac Hardware Guide »
Desktops, Laptops, Servers

See also: iPod Product Guide, iPhone Product Guide

Latest News on Macworld

Macworld 2010 refocuses for new era, without Apple

Doctors interested in Apple's iPad; 1 in 5 plan to buy one

Review: Random Play for iPhone

Panasonic introduces low-light-optimized 1080p camcorders

Developer Sourcebits puts focus on iPad apps

Disney sings the praises of the iPad to its investors

Expo: Fujitsu introduces new portable scanner

See all the latest News

event, so that Aperture doesn't attempt to match the faces in one wedding to the faces in every other photo in his or her library.

There's also an Unnamed Faces View, a film strip on the new Faces Corkboard, which shows all the faces that the program has detected but hasn't been able to name.

The implementation of Places in Aperture 3 "sets a new standard for geotagging for photographs," Paulsen said. As with iPhoto '09, there's a map interface with tiles from Google Maps and reverse-geocoding is performed by Apple's servers. But Aperture 3's map is highly interactive, allowing users to drag and drop thumbnails onto the map, move pins if the location's slightly off, and even import GPX files so that users of GPS logging tools can auto-match an entire set of photos to the GPS data. The program also writes latitude and longitude data back to the photo files' IPTC data.

Brushes and adjustments

The new Brushes feature is "probably the most requested feature in Aperture," Paulsen said. With Brushes, users can now selectively adjust portions of their images, rather than adjusting the entire image. There are 15 Quick Brushes, presets for the most common retouching tasks, such as increasing contrast, polarizing, dodging, burning, and adjusting saturation. But in addition, users can turn adjustments they've created in the Inspector window—including the new curves setting—into a brush.

Each brush stroke can be turned on or off individually, as the adjustments are entirely non-destructive. And Brushes includes an optional edge-detection feature that analyzes what you're brushing in real-time and limits adjustments to the area you intended to modify.

Aperture's adjustment-preset system has been updated. Previously, users could save an adjustment setting as a preset. Now users can take combinations of those adjustments and save them together as a single preset. Aperture 3 comes preloaded with "dozens of presets," according to Paulsen, including quick-fixes, white-balance fixes, and black-and-white and color adjustments. All presets are importable and exportable, and Paulsen said he "can see a whole ecosystem of developers selling presets, which users can buy and import."

Output and slideshows

In addition to book layout, web galleries, and web journals, Aperture now supports uploading to Facebook and Flickr. These features are "pretty much the same as iPhoto," Paulsen said, though he added that Faces names in a photo can now be added to Flickr as keywords.

Aperture 3 boasts a new slideshow engine, which offers the features of iPhoto '09's advanced slideshows—but with all the controls opened up. Users can define slide durations and transitions, add text and borders, define multilayered soundtracks, and even add video. (Aperture 3 imports and supports video, offering a trimming interface similar to iPhoto.) Slideshows can then be rendered to video and sent to iTunes, and Aperture 3 supports exporting slideshows in 720p and 1080p formats for display on HDTVs.

Other advancements

Aperture 3 offers a new full-screen view, that lets you work on projects, thumbnails, and even the Browser in full-screen view. And there's a Library Path Navigator feature that allows you "to navigate to any part of your library without leaving full-screen mode," Paulsen said.

Library handling is also improved in Aperture 3. If you've got multiple Aperture library files, you can merge them all together in one place. A photographer with a MacBook Pro on a remote photo shoot can start with a fresh library and import it into his or her master library upon returning home. More impressively, Aperture will sync and merge libraries: photographers can export a portion of their libraries to a laptop, take it on the road to do work, and then bring that library back home. Aperture will detect what's changed and sync only the changes back to the master library.

The new version of Aperture runs in 64-bit mode on Snow Leopard, and Paulsen said that's especially good news for users who have giant files, such as super high-resolution scanned images. Those images can now be loaded into memory, vastly increasing performance.

The last major update to Aperture, version 2 (), was in February 2008.

See more like this: photo editing, creativepro

"Apple releases Aperture 3" Comments

Apple releases iLife, Raw, and Aperture updates

Apple releases Aperture 3

Looking Back: Photoshop turns 20

https://web.archive.org/web/20100315070539/http://www.apple.com/aperture/resources/

Resources

Downloads

Aperture Downloads

Update to the latest version of Aperture — by visiting the Aperture Downloads page. If you're unable to upgrade to Aperture 3, updates for earlier versions of Aperture are also available on this page.

AppleScript and Automator Support

Aperture offers integrated support for AppleScript and Automator, letting you create custom workflows and automate some of the activities you perform over and over. Download the Aperture 3 AppleScript Reference for a complete guide to using AppleScript with Aperture. This 36-page document describes classes, commands, and other AppleScript-specific features found in the Aperture 3 AppleScript dictionary.

For help with Automator, third parties — like automator.us* — have created ready-to-use workflows you can download and put to work today.

* Workflows available at automator.us are provided as is as a courtesy. Apple makes no warranties about their usefulness or quality.

Third-Party Websites

Aperture Users Network

From tips to articles on the digital workflow to podcasts, Aperture users have a great source for information and inspiration in the Aperture Users Network.
http://aperture.maccreate.com

Services

Print Products

Order professional-quality lab prints or beautifully bound softcover or hardcover photo books in multiple sizes. Books and prints are competitively priced and printed to Apple's exacting quality standards. Learn more ▸

Third-Party Plug-ins, Presets, and Other Extras

Aperture 3 includes a powerful plug-in architecture for the seamless integration of popular third-party image editing and export plug-ins. These plug-ins allow you to extend the capabilities of Aperture by accessing an entire industry's worth of imaging expertise — without ever leaving Aperture. See all plug-ins ▸

- Image Editing Plug-ins. These plug-ins extend the built-in image editing capabilities of Aperture, adding specialized tools for noise reduction, selective adjustments, lens correction, and much more.
- Export Plug-ins. Streamline your work by sending your photos to Flickr, Facebook, SmugMug, and other photo-sharing websites directly from Aperture. Or easily upload your photos to a remote FTP server, send them to another application, or generate a Flash-based web gallery.
- Photo Book Plug-ins. Now you can easily create and order photo albums from some of the finest bookmakers in the world, right in Aperture. Just download a plug-in, and you're good to go.
- Automations and Scripts. These automated workflows take advantage of AppleScript to turn complex multistep tasks into one-click operations that extend existing Aperture features, and make it easy to integrate Aperture into a workflow that includes other applications such as Keynote, Mail, or InDesign.
- Extras. These third-party web themes offer a variety of creative design options that give you greater flexibility when presenting your photos on the web.

The free Imaging Plug-in Software Development Kit (SDK) for Aperture is available through the Apple Developer Connection (ADC). Information about developing your own plug-ins for Aperture 3 is available on the Apple Developer website.

Inquiries regarding the development of photo book plug-ins can be sent to aperturedeveloper@apple.com.

Support

Aperture Support Page

From software updates to technical articles to manuals to video tutorials, the Aperture support page is a handy resource with a wide variety of technical information about Aperture 3.
Learn more ▸

Aperture Forums

Discuss Aperture with other photographers eager to share information, answer questions, and offer tips and advice to fellow Aperture users. Learn more ▸

One to One

Get more out of Aperture 3 with a One to One membership at the Apple Retail Store. We'll teach you the basics of using Aperture or walk you through advanced projects — whatever you need to build your skills. One to One is available only at the time you purchase a new Mac from the Apple Retail Store or Apple Online Store.

Learn more ▸

Aperture 3 Free Trial

Take your photos further. Try Aperture free for 30 days.

From iPhoto to Aperture

All of your iPhoto Events, Faces, Places, albums, and more will be preserved.

One-of-a-Kind Photo Books

Now it's easy to create custom, professional-quality photo books.

Buy Aperture 3

Buy Now

Find an Apple Retail Store.
Find your local authorized reseller.
Get Apple education pricing.
Call 1-800-MY-APPLE.

https://web.archive.org/web/20100411022811/http://www.apple.com/aperture

Aperture 3

Taking photos. Further.

The new Aperture 3 gives you powerful yet easy-to-use tools to refine images, showcase your photography, and manage massive libraries on your Mac. It's pro performance with iPhoto simplicity. Learn more ▸

What is Aperture?

It's the way to better photos on a Mac. With tools to retouch photos, organize libraries, share work online, and print professionally designed books. Learn more ▸

New in Aperture 3

Organization with Faces and Places. Brushes and adjustment presets to perfect and enhance images. True full-screen browsing. And over 200 more new features. Learn more ▸

Aperture in Action

For his documentary project Wisdom of New York, professional photographer Doug Menuez uses Aperture 3 to manage his photos on location and sync edits back in the studio. Watch video ▸

Go from iPhoto to Aperture

You've taken some amazing shots and iPhoto has been a solid photo assistant. Now you're ready to go further. The move to Aperture 3 is designed to be seamless. Learn more ▸

200+
New Features
More power, more ease of use, and more possibilities. Learn More ▸

Buy Aperture 3 now.

Find an Apple Retail Store.
Find your local authorized reseller.
Get Apple education pricing.
Call 1-800-MY-APPLE.
View system requirements.

Download the 30-day free trial.

30-Day Free Trial

Download Free Trial

Music
iWork
MobileMe

LED Cinema Display

Safari
Learn more

Shop the Apple Online Store (1-800-MY-APPLE), visit an Apple Retail Store, or find a reseller.

Apple Info | Site Map | Hot News | RSS Feeds | Contact Us |

 Terms of Use | Privacy Policy

https://web.archive.org/web/20100311173011/http://www.apple.com:80/support/aperture/

All products...

Apple Photo Services

Aperture

Aperture 3 User Manual
Get to know the ins and outs of Aperture.

Discussions forums
Seek help from other Aperture users.

Software Update
Get the latest updates to your Aperture software.

Downloads Manuals Specifications

Get Started

- Aperture: Optimizing for your printer
- Find answers to questions about your order
- Preparing your photos for ordering

More

Troubleshooting

More

Discuss Aperture topics with other users in our customer forums.

- Brand new M1 MacBook Air, Photos is already eating battery?

See all topics

- Aperture 3.0.1
- Aperture 3 Trial
- Aperture 2.1.4

- Aperture 3 User Manual
- Exploring Aperture 3
- Aperture 3 Keyboard Shortcuts
- Release Notes
- Aperture 3 New Features
- Frequently Asked Questions
- Photography Fundamentals
- Performing Adjustments
- About Badge Overlays
- Aperture Resources
- Aperture Tutorials

- Find an Apple Store near you
- Find an Apple Authorized Service Provider

https://web.archive.org/web/20100315070539/http://www.apple.com:80/aperture/resources/

Resources

Downloads

Aperture Downloads

Update to the latest version of Aperture — by visiting the Aperture Downloads page. If you're unable to upgrade to Aperture 3, updates for earlier versions of Aperture are also available on this page.

AppleScript and Automator Support

Aperture offers integrated support for AppleScript and Automator, letting you create custom workflows and automate some of the activities you perform over and over. Download the Aperture 3 AppleScript Reference for a complete guide to using AppleScript with Aperture. This 36-page document describes classes, commands, and other AppleScript-specific features found in the Aperture 3 AppleScript dictionary.

For help with Automator, third parties — like automator.us* — have created ready-to-use workflows you can download and put to work today.

* Workflows available at automator.us are provided as is as a courtesy. Apple makes no warranties about their usefulness or quality.

Third-Party Websites

Aperture Users Network

From tips to articles on the digital workflow to podcasts, Aperture users have a great source for information and inspiration in the Aperture Users Network.
http://aperture.maccreate.com

Services

Print Products

Order professional-quality lab prints or beautifully bound softcover or hardcover photo books in multiple sizes. Books and prints are competitively priced and printed to Apple's exacting quality standards. Learn more ▸

Third-Party Plug-ins, Presets, and Other Extras

Aperture 3 includes a powerful plug-in architecture for the seamless integration of popular third-party image editing and export plug-ins. These plug-ins allow you to extend the capabilities of Aperture by accessing an entire industry's worth of imaging expertise — without ever leaving Aperture. See all plug-ins ▸

- Image Editing Plug-ins. These plug-ins extend the built-in image editing capabilities of Aperture, adding specialized tools for noise reduction, selective adjustments, lens correction, and much more.
- Export Plug-ins. Streamline your work by sending your photos to Flickr, Facebook, SmugMug, and other photo-sharing websites directly from Aperture. Or easily upload your photos to a remote FTP server, send them to another application, or generate a Flash-based web gallery.
- Photo Book Plug-ins. Now you can easily create and order photo albums from some of the finest bookmakers in the world, right in Aperture. Just download a plug-in, and you're good to go.
- Automations and Scripts. These automated workflows take advantage of AppleScript to turn complex multistep tasks into one-click operations that extend existing Aperture features, and make it easy to integrate Aperture into a workflow that includes other applications such as Keynote, Mail, or InDesign.
- Extras. These third-party web themes offer a variety of creative design options that give you greater flexibility when presenting your photos on the web.

The free Imaging Plug-in Software Development Kit (SDK) for Aperture is available through the Apple Developer Connection (ADC). Information about developing your own plug-ins for Aperture 3 is available on the Apple Developer website.

Inquiries regarding the development of photo book plug-ins can be sent to aperturedeveloper@apple.com.

Support

Aperture Support Page

From software updates to technical articles to manuals to video tutorials, the Aperture support page is a handy resource with a wide variety of technical information about Aperture 3.
Learn more ▸

Aperture Forums

Discuss Aperture with other photographers eager to share information, answer questions, and offer tips and advice to fellow Aperture users. Learn more ▸

One to One

Get more out of Aperture 3 with a One to One membership at the Apple Retail Store. We'll teach you the basics of using Aperture or walk you through advanced projects — whatever you need to build your skills. One to One is available only at the time you purchase a new Mac from the Apple Retail Store or Apple Online Store.

Learn more ▸

Aperture 3 Free Trial

Take your photos further. Try Aperture free for 30 days.

From iPhoto to Aperture

All of your iPhoto Events, Faces, Places, albums, and more will be preserved.

One-of-a-Kind Photo Books

Now it's easy to create custom, professional-quality photo books.

Buy Aperture 3

Buy Now

Find an Apple Retail Store.
Find your local authorized reseller.
Get Apple education pricing.
Call 1-800-MY-APPLE.

https://web.archive.org/web/20100316213353/http://www.apple.com/aperture/

Aperture 3

Taking photos. Further.

The new Aperture 3 gives you powerful yet easy-to-use tools to refine images, showcase your photography, and manage massive libraries on your Mac. It's pro performance with iPhoto simplicity. Learn more ▸

What is Aperture?

It's the way to better photos on a Mac. With tools to retouch photos, organize libraries, share work online, and print professionally designed books. Learn more ▸

New in Aperture 3

Organization with Faces and Places. Brushes and adjustment presets to perfect and enhance images. True full-screen browsing. And over 200 more new features. Learn more ▸

Aperture in Action

With Aperture 3, photojournalist Bill Frakes creates a multimedia slideshow that combines still photos, audio, and HD video to document people's everyday lives. Watch video ▸

Go from iPhoto to Aperture

You've taken some amazing shots and iPhoto has been a solid photo assistant. Now you're ready to go further. The move to Aperture 3 is designed to be seamless. Learn more ▸

200+ New Features

More power, more ease of use, and more possibilities. Learn More ▸

Buy Aperture 3 now.

Find an Apple Retail Store.
Find your local authorized reseller.
Get Apple education pricing.
Call 1-800-MY-APPLE.
View system requirements.

Download the 30-day free trial.

30-Day Free Trial

Download Free Trial

Music
iWork
MobileMe

LED Cinema Display

Safari
Learn more

Shop the Apple Online Store (1-800-MY-APPLE), visit an Apple Retail Store, or find a reseller.

Apple Info | Site Map | Hot News | RSS Feeds | Contact Us |

 Terms of Use | Privacy Policy

https://web.archive.org/web/20100114012530/http://developer.apple.com/mac/library/documentation/cocoa/Conceptual/LoadingResources/ImageSoundResources/ImageSoundResources.html#//apple_ref/doc/uid/10000051i-CH7-SW1

Image, Sound, and Video Resources

The Mac OS X and iPhone OS platforms were built to provide a rich multimedia experience. To support that experience, both platforms provide plenty of support for loading and using image, sound, and video resources in your application. Image resources are commonly used to draw portions of an application's user interface. Sound and video resources are used less frequently but can also enhance the basic appearance and appeal of an application. The following sections describe the support available for working with image, sound, and video resources in your applications.

Images and Sounds in Nib Files

Using Interface Builder, you can reference your application's sound and image files from within nib files. You might do so to associate those images or sounds with different properties of a view or control. For example, you might set the default image to display in an image view or set the image to display for a button. Creating such a connection in Interface Builder saves you the hassle of having to make that connection later when the nib file is loaded.

To make image and sound resources available in Interface Builder 3.0 and later, all you have to do is add them to your Xcode project. Interface Builder automatically searches your Xcode project for resources and lists them in the library window. When you make a connection to a given resource file, Interface Builder makes a note of that connection in the nib file. At load time, the nib-loading code looks for that resource in the project bundle, where it should have been placed by Xcode at build time.

When you load a nib file that contains references to image and sound resources, the nib-loading code caches resources whenever possible for easy retrieval later. For example, after loading a nib file, you can retrieve an image associated with that nib file using the `imageNamed:` method of either `NSImage` or `UIImage` (depending on your platform). Similarly, you can retrieve cached sound resources in Mac OS X using the `soundNamed:` method of `NSSound`. Interface Builder does not cache sound resources in iPhone OS nor does it cache image and sound resources associated with Carbon nib files.

Loading Image Resources

Image resources are commonly used in most applications. Even very simple applications use images to create a custom look for controls and views. Mac OS X and iPhone OS provide extensive support for manipulating image data using Objective-C objects. These objects make using image images extremely easy, often requiring only a few lines of code to load and draw the image. If you prefer not to use the Objective-C objects, you can also use Quartz to load images using a C-based interface. The following sections describe the process for loading image resource files using each of the available techniques.

Loading Images in Objective-C

To load images in Objective-C, you use either the `NSImage` or `UIImage` object, depending on the current platform. Applications built for Mac OS X using the AppKit framework use the `NSImage` object to load images and draw them. Applications built for iPhone OS use the `UIImage` object. Functionally, both of these objects provide almost identical behavior when it comes to loading existing image resources. You initialize the object by passing it a pointer to the image file in your application bundle and the image object takes care of the details of loading and drawing the image data.

Listing 4-1 shows how to load an image resource using the `NSImage` class. After you locate the image resource, which in this case is in the application bundle, you simply use that path to initialize the image object. After initialization, you can draw the image using the methods of `NSImage` or pass that object to other methods that can use it. To perform the exact same task in iPhone OS, all you would need to do is change references of `NSImage` to `UIImage`.

Listing 4-1 Loading an image resource

```
NSString* imageName = [[NSBundle mainBundle] pathForResource:@"image1" ofType:@"png"];

NSImage* imageObj = [[NSImage alloc] initWithContentsOfFile:imageName];
```

You can use image objects to open any type of image supported on the target platform. Each object is typically a lightweight wrapper for more advanced image handling code. To draw an image in the current graphics context, you would simply use one of its drawing related methods. Both `NSImage` and `UIImage` have methods for drawing the image in several different ways. The `NSImage` class also provides extra support for manipulating the images you load.

For information about the methods of the `NSImage` and `UIImage` classes, see *NSImage Class Reference* and *UIImage Class Reference*. For more detailed information about the additional features of the `NSImage` class, see Images in *Cocoa Drawing Guide*.

Loading Images Using Quartz

If you are writing C-based code, you can use a combination of Core Foundation and Quartz calls to load image resources into your applications. Core Foundation provides the initial support for locating image resources and loading the corresponding image data into memory. Quartz takes the image data you load into memory and turns it into a usable `CGImageRef` that your code can then use to draw the image.

There are two ways to load images using Quartz: data providers and image source objects. Data providers are available in both iPhone OS and Mac OS X. Image source objects are available only in Mac OS X v10.4 and later but take advantage of the Image I/O framework to enhance the basic image handling capabilities of data providers. When it comes to loading and displaying image resources, both technologies are well suited for the job. The only time you might prefer image sources over data providers is when you want greater access to the image-related data.

Listing 4-2 shows how to use a data provider to load a JPEG image. This method uses the Core Foundation bundle support to locate the image in the application's main bundle and get a URL to it. It then uses that URL to create the data provider object and then create a `CGImageRef` for the corresponding JPEG data. (For brevity this example omits any error-handling code. Your own code should make sure that any referenced data structures are valid.)

Listing 4-2 Using data providers to load image resources

```
CGImageRef MyCreateJPEGImageRef (const char *imageName);
```

```
{

    CGImageRef image;

    CGDataProviderRef provider;

    CFStringRef name;

    CFURLRef url;

    CFBundleRef mainBundle = CFBundleGetMainBundle();



    // Get the URL to the bundle resource.

    name = CFStringCreateWithCString (NULL, imageName, kCFStringEncodingUTF8);

    url = CFBundleCopyResourceURL(mainBundle, name, CFSTR("jpg"), NULL);

    CFRelease(name);



    // Create the data provider object

    provider = CGDataProviderCreateWithURL (url);

    CFRelease (url);



    // Create the image object from that provider.

    image = CGImageCreateWithJPEGDataProvider (provider, NULL, true,

                                    kCGRenderingIntentDefault);

    CGDataProviderRelease (provider);



    return (image);

}
```

For detailed information about working with Quartz images, see *Quartz 2D Programming Guide*. For reference information about data providers, see *Quartz 2D Reference Collection* (Mac OS X) or *Core Graphics Framework Reference* (iPhone OS).

Playing Audio Files

Audio resources are typically used to provide audio feedback for different parts of your application. Several technologies are available to handle the loading and playback of audio. Which technology you use is going to be determined by the underlying platform and the level of sophistication you need for handling the audio. The following sections describe the key technologies you might use and when you would use them.

Using Core Audio to Play Sounds

Both Mac OS X and iPhone OS support the playback of audio files using the Core Audio family of frameworks. Core Audio provides a wide range of audio services, including the playback of essentially any kind of audio file you can imagine. For basic playback, Core Audio offers two mechanisms, both available in the Audio Toolbox framework:

- To play short sound files of under five seconds duration when you do not need level control or other control, use System Audio Services.
- To play longer sound files, to exert control over playback including level adjustments, or to play multiple sounds simultaneously, use Audio Queue Services.

Listing 4-3 shows a short program that uses the interfaces in System Audio Services to play a sound. Before playing the sound, it registers it and creates a sound ID for it. To play the sound, it then passes this sound ID to the `AudioServicesPlaySystemSound` function. When the sound is finished playing, Core Audio notifies the application by calling its audio completion callback routine. This routine handles the clean up of the sound ID prior to the program exiting.

Listing 4-3 Playing a sound using System Audio Services

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

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


// Define a callback to be called when the sound is finished
// playing. Useful when you need to free memory after playing.
static void MyCompletionCallback (
    SystemSoundID  mySSID,
    void * myURLRef
) {
        AudioServicesDisposeSystemSoundID (mySSID);
        CFRelease (myURLRef);
        CFRunLoopStop (CFRunLoopGetCurrent());
}


int main (int argc, const char * argv[]) {
    // Set up the pieces needed to play a sound.
    SystemSoundID    mySSID;
    CFURLRef        myURLRef;
    myURLRef = CFURLCreateWithFileSystemPath (
        kCFAllocatorDefault,
        CFSTR ("../../ComedyHorns.aif"),
        kCFURLPOSIXPathStyle,
        FALSE
    );


    // create a system sound ID to represent the sound file
    OSStatus error = AudioServicesCreateSystemSoundID (myURLRef, &mySSID);


    // Register the sound completion callback.
    // Again, useful when you need to free memory after playing.
    AudioServicesAddSystemSoundCompletion (
        mySSID,
        NULL,
        NULL,
        MyCompletionCallback,
```

```
        (void *) myURLRef

    );



    // Play the sound file.

    AudioServicesPlaySystemSound (mySSID);



    // Invoke a run loop on the current thread to keep the application

    // running long enough for the sound to play; the sound completion

    // callback later stops this run loop.

    CFRunLoopRun ();

    return 0;

}
```

For more information about the features of Core Audio, see *Core Audio Overview*. For information and examples of how to play sounds using the Audio Queue Services technology, see *Audio Queue Services Programming Guide*.

Using the AppKit Framework to Play Audio

In Mac OS X, the AppKit framework provides support for loading and playing sound files through the `NSSound` class. You can use this class to play back sounds stored as AIFF, WAV, and NeXT `.snd` files. For sound resources located in your application's bundle, the simplest way to load a sound is using the `soundNamed:` method, as shown in the following example:

```
NSSound* aSound = [NSSound soundNamed:@"mySound"];
```

The `soundNamed:` method checks the application's sound cache for an existing sound resource with the specified name. If the specified resource is not currently in the sound cache, `NSSound` automatically searches for it in several other locations, including your application's main bundle and any system `Library/Sounds` directories.

Because the `soundNamed:` method also loads system sound names, you should avoid using the names of system sounds when naming any of your custom sound files. Cocoa populates the sound cache with any sound files it needs, such as the file used for the current system alert sound. It caches these sounds under the filename of the sound (minus its filename extension). If one of your custom sounds matches the name of a different sound file that is already cached, the `soundNamed:` method returns the cached file instead of your custom one.

If you want to ensure that the correct sound file resource is loaded every time, you can always load the sound file using an explicit path string, as shown in the following example.

```
NSString* soundFile = [[NSBundle mainBundle] pathForResource:@"mySound" ofType:@"aiff"];

NSSound* sound = [[NSSound alloc] initWithContentsOfFile:soundFile byReference:YES];
```

Note: Sound files associated with a nib file are loaded automatically when the nib file is loaded. To access those sounds, use the `soundNamed:` method of `NSSound`, passing in the name of the sound. For more information, see "About Image and Sound Resources."

For more information about using the `NSSound` class, see *Sound Programming Topics for Cocoa* and *NSSound Class Reference*. If you want to load sound resources for Carbon-based applications, you must use QuickTime or Core Audio to do so. For information about the QuickTime Kit framework, see *QuickTime Kit Programming Guide* and *QTKit Framework Reference*. For general information about the QuickTime framework, see *QuickTime Overview*.

Playing Video Resources

Video resources are prerendered movie files that you can play from your application's user interface. Games often use prerendered movies as cut scenes between different levels. The following sections provide information about how to load these types of resources and play them in your applications.

Playing Video Files in Mac OS X

Video files are like any other resource files in your application. Once you locate the resource file, you can use an appropriate technology to open and play it. In Mac OS X, you use the QuickTime or Quicktime Kit frameworks to open video files, associate them with a graphics context, and play their contents. These frameworks support the playback of both video and audio files in either C or Objective-C code.

The following example loads a video file from an application's bundle and associates it with a view using the QuickTime Kit framework. The view object returned by the `getMyQTMovieView` method is assumed to be a `QTMovieView` object located in one of the caller's windows.

```
NSString*   movieFile = [[NSBundle mainBundle] pathForResource:@"myMovie" ofType:@"mov"];

QTMovie*    aMovie = [QTMovie movieWithFile:movieFile error:nil];
```

```
// Install the movie in a custom movie view associated with the caller.

QTMovieView*   myView = [self getMyQTMovieView];

[myView setMovie:aMovie];
```

Prior to Mac OS X v10.4, you can use the `NSMovie` and `NSMovieView` classes in Cocoa to load and display video files. In Mac OS X v10.4 and later, it is recommended that you use the classes of the QuickTime Kit framework instead.

For C-based applications, you can load video files using either the QuickTime framework or the QuickTime Kit framework. If you choose to use the QuickTime Kit framework, you must incorporate Objective-C code into your project. For information on how to use Objective-C code in Carbon applications, see *Carbon-Cocoa Integration Guide*.

For information about the QuickTime Kit framework, see *QuickTime Kit Programming Guide* and *QTKit Framework Reference*. For general information about the QuickTime framework, see *QuickTime Overview*. For details of how to incorporate movie content into your application, see *QuickTime Movie Basics*.

Playing Video Files in iPhone OS

iPhone OS supports the ability to play back video files directly from your application using the Media Player framework (`MediaPlayer.framework`). Video playback is supported in full screen mode only and can be used by game developers who want to play cut scene animations or by other developers who want to play media files. When you start a video from your application, the media player interface takes over, fading the screen to black and then fading in the video content. You can play a video with or without transport controls; enabling transport controls lets the user pause or adjust the playback of the video. If you do not enable these controls, the video plays until completion or until you explicitly stop it in your code.

To initiate video playback, you must know the URL of the file you want to play. For files your application provides, this would typically be a pointer to a file in your application's bundle; however, it can also be a pointer to a file on a remote server or elsewhere in the directory containing your application. You use this URL to instantiate a new instance of the `MPMoviePlayerController` class. This class presides over the playback of your video file and manages user interactions, such user taps in the transport controls (if shown). To initiate playback, simply call the `play` method of the controller.

Listing 4-4 shows a sample method that playbacks the video at the specified URL. The play method is an asynchronous call that returns control to the caller while the movie plays. The movie controller loads the movie in a full-screen view, and animates the movie into place on top of the application's existing content. When playback is finished, the movie controller sends a notification to the object, which releases the movie controller now that it is no longer needed.

Listing 4-4 Playing full screen movies.

```
-(void)playMovieAtURL:(NSURL*)theURL

{

    MPMoviePlayerController* thePlayer = [[MPMoviePlayerController alloc] initWithContentURL:theURL];



    thePlayer.scalingMode = MPMovieScalingModeAspectFill;

    thePlayer.userCanShowTransportControls = NO;



    // Register for the playback finished notification.

    [[NSNotificationCenter defaultCenter] addObserver:self

                selector:@selector(myMovieFinishedCallback:)

                name:MPMoviePlayerPlaybackDidFinishNotification

                thePlayer];



    // Movie playback is asynchronous, so this method returns immediately.

    [thePlayer play];

}



// When the movie is done, release the controller.

-(void)myMovieFinishedCallback:(NSNotification*)aNotification

{
```

```
{

    MPMoviePlayerController* thePlayer = [aNotification object];



    [[NSNotificationCenter defaultCenter] removeObserver:self

                name:MPMoviePlayerPlaybackDidFinishNotification

                thePlayer];



    // Release the movie instance created in playMovieAtURL:

    [thePlayer release];

}
```

Last updated: 2009-01-06

Did this document help you? Yes It's good, but... Not helpful...

https://web.archive.org/web/20100112225732/http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/LoadingResources/Strings/Strings.html#//apple_ref/doc/uid/10000051i-CH6-SW1

String Resources

An important part of the localization process is to localize all of the text strings displayed by your application. By their nature, strings located in nib files can be readily localized along with the rest of the nib file contents. Strings embedded in your code, however, must be extracted, localized, and then reinserted back into your code. To simplify this process—and to make the maintenance of your code easier—Mac OS X and iPhone OS provide the infrastructure needed to separate strings from your code and place them into resource files where they can be localized easily.

Resource files that contain localizable strings are referred to as **strings** files because of their filename extension, which is `.strings`. You can create strings files manually or programmatically depending on your needs. The standard strings file format consists of one or more key-value pairs along with optional comments. The key and value in a given pair are strings of text enclosed in double quotation marks and separated by an equal sign. (You can also use a property list format for strings files. In such a case, the top-level node is a dictionary and each key-value pair of that dictionary is a string entry.)

Listing 3-1 shows a simple strings file that contains non-localized entries for the default language. When you need to display a string, you pass the string on the left to one of the available string-loading routines. What you get back is the matching value string containing the text translation that is most appropriate for the current user. For the development language, it is common to use the same string for both the key and value, but doing so is not required.

Listing 3-1 A simple strings file

```
/* Insert Element menu item */

"Insert Element" = "Insert Element";

/* Error string used for unknown error types. */

"ErrorString_1" = "An unknown error occurred.";
```

A typical application has at least one strings file per localization, that is, one strings file in each of the bundle's `.lproj` subdirectories. The name of the default strings file is `Localizable.strings` but you can create strings files with any file name you choose. Creating strings files is discussed in more depth in "Creating Strings Resource Files."

Note: It is recommended that you save strings files using the UTF-16 encoding, which is the default encoding for standard strings files. It is possible to create strings files using other property-list formats, including binary property-list formats and XML formats that use the UTF-8 encoding, but doing so is not recommended. For more information about the standard strings file format, see "Creating Strings Resource Files." For more information about Unicode and its text encodings, go to http://www.unicode.org/ or http://en.wikipedia.org/wiki/Unicode.

The loading of string resources (both localized and nonlocalized) ultimately relies on the bundle and internationalization support found in both Mac OS X and iPhone OS. For information about bundles, see *Bundle Programming Guide*. For more information about internationalization and localization, see *Internationalization Programming Topics*.

Creating Strings Resource Files

Although you can create strings files manually, it is rarely necessary to do so. The easiest way to create strings files is to write your code using the appropriate string-loading macros and then use the `genstrings` command-line tool to extract those strings and create strings files for you.

The following sections describe the process of how to set up your source files to facilitate the use of the `genstrings` tool. For detailed information about the tool, see `genstrings` man page.

Choosing Which Strings to Localize

When it comes to localizing your application's interface, it is not always appropriate to localize every string used by your application. Translation is a costly process, and translating strings that are never seen by the user is a waste of time and money. Strings that are not displayed to the user, such as notification names used internally by your application, do not need to be translated. Consider the following example:

```
if (CFStringHasPrefix(value, CFSTR("-")) {    CFArrayAppendValue(myArray, value);};
```

In this example, the string "-" is used internally and is never seen by the user; therefore, it does not need to be placed in a strings file.

The following code shows another example of a string the user would not see. The string `"%d %d %s"` does not need to be localized, since the user never sees it and it has no effect on anything that the user does see.

```
matches = sscanf(s, "%d %d %s", &first, &last, &other);
```

Because nib files are localized separately, you do not need to include strings that are already located inside of a nib file. Some of the strings you should localize, however, include the following:

- Strings that are programmatically added to a window, panel, view, or control and subsequently displayed to the user. This includes strings you pass into standard routines, such as those that display alert boxes.
- Menu item title strings if those strings are added programmatically. For example, if you use custom strings for the Undo menu item, those strings should be in a strings file.
- Error messages that are displayed to the user.

- Any boilerplate text that is displayed to the user.
- Some strings from your application's information property list (`Info.plist`) file; see *Runtime Configuration Guidelines*.
- New file and document names.

About the String-Loading Macros

The Foundation and Core Foundation frameworks define the following macros to make loading strings from a strings file easier:

- Core Foundation macros:
  - `CFCopyLocalizedString`
  - `CFCopyLocalizedStringFromTable`
  - `CFCopyLocalizedStringFromTableInBundle`
  - `CFCopyLocalizedStringWithDefaultValue`
- Foundation macros:
  - `NSLocalizedString`
  - `NSLocalizedStringFromTable`
  - `NSLocalizedStringFromTableInBundle`
  - `NSLocalizedStringWithDefaultValue`

You use these macros in your source code to load strings from one of your application's strings files. The macros take the user's current language preferences into account when retrieving the actual string value. In addition, the `genstrings` tool searches for these macros and uses the information they contain to build the initial set of strings files for your application.

For detailed information about how to use these macros, see "Loading String Resources Into Your Code."

Using the Genstrings Tool to Create Strings Files

At some point during your development, you need to create the strings files needed by your code. If you wrote your code using the Core Foundation and Foundation macros, the simplest way to create your strings files is using the `genstrings` command-line tool. You can use this tool to generate a new set of strings files or update a set of existing files based on your source code.

To use the `genstrings` tool, you typically provide at least two arguments:

- A list of source files
- An optional output directory

The `genstrings` tool can parse C, Objective-C, and Java code files with the `.c`, `.m`, or `.java` filename extensions. Although not strictly required, specifying an output directory is recommended and is where `genstrings` places the resulting strings files. In most cases, you would want to specify the directory containing the project resources for your development language.

The following example shows a simple command for running the `genstrings` tool. This command causes the tool to parse all Objective-C source files in the current directory and put the resulting strings files in the `en.lproj` subdirectory, which must already exist.

```
genstrings -o en.lproj *.m
```

The first time you run the `genstrings` tool, it creates a set of new strings files for you. Subsequent runs replace the contents of those strings files with the current string entries found in your source code. For subsequent runs, it is a good idea to save a copy of your current strings files before running `genstrings`. You can then diff the new and old versions to determine which strings were added to (or changed in) your project. You can then use this information to update any already localized versions of your strings files, rather than replacing those files and localizing them again.

Within a single strings file, each key must be unique. Fortunately, the `genstrings` tool is smart enough to coalesce any duplicate entries it finds. When it discovers a key string used more than once in a single strings file, the tool merges the comments from the individual entries into one comment string and generates a warning. (You can suppress the duplicate entries warning with the `-q` option.) If the same key string is assigned to strings in different strings files, no warning is generated.

For more information about using the `genstrings` tool, see the `genstrings` man page.

Creating Strings Files Manually

Although the `genstrings` tool is the most convenient way to create strings files, you can also create them manually. To create a strings file manually, create a new file in TextEdit (or your preferred text-editing application) and save it using the Unicode UTF-16 encoding. (When saving files, TextEdit usually chooses an appropriate encoding by default. To force a specific encoding, you must change the save options in the application preferences.) The contents of this file consists of a set of key-value pairs along with optional comments describing the purpose of each key-value pair. Key and value strings are separated by an equal sign, and the entire entry must be terminated with a semicolon character. By convention, comments are enclosed inside C-style comment delimiters (`/*` and `*/`) and are placed immediately before the entry they describe.

Listing 3-2 shows the basic format of a strings file. The entries in this example come from the English version of the `Localizable.strings` file from the TextEdit application. The left side of each equal sign represents the key, and the right side represents the value. A common convention when developing applications is to use a key name that equals the value in the language used to develop the application. Therefore, because TextEdit was developed using the English language, the English version of the `Localizable.strings` file has keys and values that match.

Listing 3-2 Strings localized for English

```
/* Menu item to make the current document plain text */
```

```
"Make Plain Text" = "Make Plain Text";

/* Menu item to make the current document rich text */

"Make Rich Text" = "Make Rich Text";
```

Listing 3-3 shows the German translation of the same entries. These entries also live inside a file called `Localizable.strings`, but this version of the file is located in the German language project directory of the TextEdit application. Notice that the keys are still in English, but the values assigned to those keys are in German. This is because the key strings are never seen by end users. They are used by the code to retrieve the corresponding value string, which in this case is in German.

Listing 3-3 Strings localized for German

```
/* Menu item to make the current document plain text */

"Make Plain Text" = "In reinen Text umwandeln";

/* Menu item to make the current document rich text */

"Make Rich Text" = "In formatierten Text umwandeln";
```

Detecting Nonlocalizable Strings

AppKit–based applications can take advantage of built-in support to detect strings that do not need to be localized and those that need to be localized but currently are not. To use this built-in support, you must launch your application from the command line. In addition to entering the path to your executable, you must also include the name of the desired setting along with a Boolean value to indicate whether the setting should be enabled or disabled. The available settings are as follows:

- The `NSShowNonLocalizableStrings` setting identifies strings that are not localizable. The strings are logged to the shell in upper case. This option occasionally generates some false positives but is still useful overall.
- The `NSShowNonLocalizedStrings` setting locates strings that were meant to be localized but could not be found in the application's existing strings files. You can use this setting to catch problems with out-of-date localizations.

For example, to use the `NSShowNonLocalizedStrings` setting with the TextEdit application, you would enter the following in Terminal:

```
/Applications/TextEdit.app/Contents/MacOS/TextEdit -NSShowNonLocalizedStrings YES
```

Loading String Resources Into Your Code

The Core Foundation and Foundation frameworks provide macros for retrieving both localized and nonlocalized strings stored in strings files. Although the main purpose of these macros is to load strings at runtime, they also serve a secondary purpose by acting as markers that the `genstrings` tool can use to locate your application's string resources. It is this second purpose that explains why many of the macros let you specify much more information than would normally be required for loading a string. The `genstrings` tool uses the information you provide to create or update your application's strings files automatically. Table 3-1 lists the types of information you can specify for these routines and describes how that information is used by the `genstrings` tool.

Table 3-1 Common parameters found in string-loading routines

Parameter	Description
Key	The string used to look up the corresponding value. This string must not contain any characters from the extended ASCII character set, which includes accented versions of ASCII characters. If you want the initial value string to contain extended ASCII characters, use a routine that lets you specify a default value parameter. (For information about the extended ASCII character set, see the corresponding Wikipedia entry.)
Table name	The name of the strings file in which the specified key is located. The `genstrings` tool interprets this parameter as the name of the strings file in which the string should be placed. If no table name is provided, the string is placed in the default `Localizable.strings` file. (When specifying a value for this parameter, include the filename without the `.strings` extension.)
Default value	The default value to associate with a given key. If no default value is specified, the `genstrings` tool uses the key string as the initial value. Default value strings may contain extended ASCII characters.
Comment	Translation comments to include with the string. You can use comments to provide clues to the translation team about how a given string is used. The `genstrings` tool puts these comments in the strings file and encloses them in C-style comment delimiters (`/*` and `*/`) immediately above the associated entry.
Bundle	An `NSBundle` object or `CFBundleRef` type corresponding to the bundle containing the strings file. You can use this to load strings from bundles other than your application's main bundle. For example, you might use this to load localized strings from a framework or plug-in.

When you request a string from a strings file, the string that is returned depends on the available localizations (if any). The Cocoa and Core Foundation macros use the built-in bundle support to retrieve the string whose localization matches the user's current language preferences. As long as your localized resource files are placed in the appropriate language-specific project directories, loading a string with these macros should yield the appropriate string automatically. If no appropriate localized string resource is found, the bundle's loading code automatically chooses the appropriate nonlocalized string instead.

For information about internationalization in general and how to create language-specific project directories, see *Internationalization Programming Topics*. For information about the bundle structure and how resource files are chosen from a bundle directory, see *Bundle Programming Guide*.

Using the Core Foundation Framework

The Core Foundation framework defines a single function and several macros for loading localized strings from your application bundle. The `CFBundleCopyLocalizedString` function provides the basic implementation for retrieving the strings. However, it is recommended that you use the following macros instead:

- `CFCopyLocalizedString(key, comment)`
- `CFCopyLocalizedStringFromTable(key, tableName, comment)`
- `CFCopyLocalizedStringFromTableInBundle(key, tableName, bundle, comment)`
- `CFCopyLocalizedStringWithDefaultValue(key, tableName, bundle, value, comment)`

There are several reasons to use the macros instead of the `CFBundleCopyLocalizedString` function. First, the macros are easier to use for certain common cases. Second, the macros let you associate a comment string with the string entry. Third, the macros are recognized by the `genstrings` tool but the `CFBundleCopyLocalizedString` function is not.

For additional information on how to use these macros, see Working With Localized Strings in *Bundle Programming Guide*. For macro and function syntax, see *CFBundle Reference*.

Using the Foundation Framework

The Foundation framework defines a single method and several macros for loading string resources. The `localizedStringForKey:value:table:` method of the `NSBundle` class loads the specified string resource from a strings file residing in the current bundle. Cocoa also defines the following macros for getting localized strings:

- `NSLocalizedString(key, comment)`
- `NSLocalizedStringFromTable(key, tableName, comment)`
- `NSLocalizedStringFromTableInBundle(key, tableName, bundle, comment)`
- `NSLocalizedStringWithDefaultValue(key, tableName, bundle, value, comment)`

As with Core Foundation, Apple recommends that you use the Cocoa convenience macros for loading strings. The main advantage to these macros is that they can be parsed by the `genstrings` tool and used to create your application's strings files. They are also simpler to use and let you associate translation comments with each entry.

For information about the syntax of the preceding macros, see *Foundation Functions Reference*. Additional methods for loading strings are also defined in *NSBundle Class Reference*.

Examples of Getting Strings

The following examples demonstrate the basic techniques for using the Foundation and Core Foundation macros to retrieve strings. Each example assumes that the current bundle contains a strings file with the name `Custom.strings` that has been translated into French. This translated file includes the following strings:

```
/* A comment */

"Yes" = "Oui";

"The same text in English" = "Le même texte en anglais";
```

Using the Foundation framework, you can get the value of the "`Yes`" string using the `NSLocalizedStringFromTable` macro, as shown in the following example:

```
NSString* theString;

theString = NSLocalizedStringFromTable (@"Yes", @"Custom", @"A comment");
```

Using the Core Foundation framework, you could get the same string using the `CFCopyLocalizedStringFromTable` macro, as shown in this example:

```
CFStringRef theString;

theString = CFCopyLocalizedStringFromTable(CFSTR("Yes"), CFSTR("Custom"), "A comment");
```

In both examples, the code specifies the key to retrieve, which is the string "Yes". They also specify the strings file (or table) in which to look for the key, which in this case is the `Custom.strings` file. During string retrieval, the comment string is ignored.

Advanced Strings File Tips

The following sections provide some additional tips for working with strings files and string resources.

Searching for Custom Functions With genstrings

The `genstrings` tool searches for the Core Foundation and Foundation string macros by default. It uses the information in these macros to create the string entries in your project's strings files. You can also direct `genstrings` to look for custom string-loading functions in your code and use those functions in addition to the standard macros. You might use custom functions to wrap the built-in string-loading routines and perform some extra processing or you might replace the default string handling behavior with your own custom model.

If you want to use `genstrings` with your own custom functions, your functions must use the naming and formatting conventions used by the Foundation macros. The parameters for your functions must match the parameters for the corresponding macros exactly. When you invoke genstrings, you specify the `-s` option followed by the name of the function that

corresponds to the `NSLocalizedString` macro. Your other function names should then build from this base name. For example, if you specified the function name `MyStringFunction`, your other function names should be `MyStringFunctionFromTable`, `MyStringFunctionFromTableInBundle`, and `MyStringFunctionWithDefaultValue`. The `genstrings` tool looks for these functions and uses them to build the corresponding strings files.

Formatting String Resources

For some strings, you may not want to (or be able to) encode the entire string in a string resource because portions of the string might change at runtime. For example, if a string contains the name of a user document, you need to be able to insert that document name into the string dynamically. When creating your string resources, you can use any of the formatting characters you would normally use for handling string replacement in the Foundation and Core Foundation frameworks. "Note" shows several string resources that use basic formatting characters:

Listing 3-4 Strings with formatting characters

```
"Windows must have at least %d columns and %d rows." =

"Les fenêtres doivent être composes au minimum de %d colonnes et %d lignes.";

"File %@ not found." = "Le fichier %@ n'existe pas.";
```

To replace formatting characters with actual values, you use the `stringWithFormat:` method of `NSString` or the `CFStringCreateWithFormat` function, using the string resource as the format string. Foundation and Core Foundation support most of the standard formatting characters used in `printf` statements. In addition, you can use the `%@` specifier shown in the preceding example to insert the descriptive text associated with arbitrary Objective-C objects. See Formatting String Objects in *String Programming Guide for Cocoa* for the complete list of specifiers.

One problem that often occurs during translation is that the translator may need to reorder parameters inside translated strings to account for differences in the source and target languages. If a string contains multiple arguments, the translator can insert special tags of the form *n*$ (where *n* specifies the position of the original argument) in between the formatting characters. These tags let the translator reorder the arguments that appear in the original string. The following example shows a string whose two arguments are reversed in the translated string:

```
/* Message in alert dialog when something fails */

"%@ Error! %@ failed!" = "%2$@ blah blah, %1$@ blah!";
```

Using Special Characters in String Resources

Just as in C, some characters must be prefixed with a backslash before you can include them in the string. These characters include double quotation marks, the backslash character itself, and special control characters such as linefeed (`\n`) and carriage returns (`\r`).

```
"File \"%@\" cannot be opened" = " ... ";

"Type \"OK\" when done" = " ... ";
```

You can include arbitrary Unicode characters in a value string by specifying `\\U` followed immediately by up to four hexadecimal digits. The four digits denote the entry for the desired Unicode character; for example, the space character is represented by hexadecimal 20 and thus would be `\\U0020` when specified as a Unicode character. This option is useful if a string must include Unicode characters that for some reason cannot be typed. If you use this option, you must also pass the `-u` option to `genstrings` in order for the hexadecimal digits to be interpreted correctly in the resulting strings file. The `genstrings` tool assumes your strings are low-ASCII by default and only interprets backslash sequences if the `-u` option is specified.

Note: The `genstrings` tool always generates strings files using the UTF-16 encoding. If you include Unicode characters in your strings and do not use `genstrings` to create your strings files, be sure to save your strings files in the UTF-16 encoding.

Debugging Strings Files

If you run into problems during testing and find that the functions and macros for retrieving strings are always returning the same key (as opposed to the translated value), run the `/usr/bin/plutil` tool on your strings file. A strings file is essentially a property-list file formatted in a special way. Running `plutil` with the `-lint` option can uncover hidden characters or other errors that are preventing strings from being retrieved correctly.

https://web.archive.org/web/20100113155803/http://developer.apple.com/mac/library/documentation/cocoa/Conceptual/LoadingResources/CarbonNibs/CarbonNibs.html#//apple_ref/doc/uid/10000051i-CH5-SW1

Carbon Resources

Carbon applications use all of the same string, image, sound, and video resources that Cocoa applications use. Carbon applications also support the use of nib files for loading the application's user interface. Unlike Cocoa applications, however, Carbon applications typically do not instantiate all of the objects in a nib file at once. Instead, applications load individual windows and menus from one or more nib files at different points in the program. This chapter provides examples of how to load nib files containing Carbon objects.

Note: Carbon supports the loading of nib files through Interface Builder Services, which is part of the HIToolbox in the Carbon framework (`Carbon.framework`). For information about the functions in Interface Builder Services, see *Interface Builder Services Reference*.

Nib File Design Guidelines

When creating your Carbon nib files, keep the following guidelines in mind:

- Store only launch-related resources in your application's main nib file. The main nib file should contain the application menu bar and any resources that are needed at launch time only. Other resources should be stored separately.
- Store frequently used user-interface components (such as document windows) in their own separate nib files. This nib file should contain only the resources needed immediately by that component.
- Store infrequently used user-interface components (such as alert panels) in separate nib files. Avoid storing infrequently used components in nib files with frequently used components. When a nib file is loaded, the data for the entire file is loaded into memory and objects are instantiated from that data as needed. Thus, if an object is not used it is still occupying space in memory.

Loading Objects from Nib Files

There are three main ways to load objects. You can:

- Load a menu bar and main window from your application's main nib file.
- Load objects from an auxiliary nib file in the main bundle.
- Load objects from nib files in other bundles.

In this section, you'll find steps and code examples for each technique.

Unarchiving Objects from the Main Nib File

When your application starts up, you need to call Interface Builder Services functions to open the main nib file and unarchive the interface objects that should be open after startup. The main nib file should contain only those items that are essential when your application starts up. In most cases, the main nib file should contain only the menu bar and perhaps a main window.

The steps below outline how to open a main nib file and unarchive the objects in it. Listing 5-1 shows how to implement the steps for a nib file that contains a menu bar and a main window. If your application needs only one of these objects at startup, you can easily modify the sample code.

1. Call the function `CreateNibReference` to create a reference to the main nib file.
2. Unarchive the menu bar from the main nib file by calling the function `SetMenuBarFromNib`. This function also sets the menu bar so users can use the menu bar when your application has started up.
3. If your application has a main window, call the function `CreateWindowFromNib` to unarchive the main window.
4. After you have unarchived the objects from the main nib file, dispose of the nib reference by calling the function `DisposeNibReference`.
5. The function `CreateWindowFromNib` unarchives the window so that it is hidden. If you want the window to be visible, you must call the Window Manager function `ShowWindow`.

It is good practice to check for errors each step of the way, as shown in Listing 5-1. If the main user interface cannot be created, your application should halt the start up process and exit. This example assumes that the name of your application's main nib file is `main.nib`.

Listing 5-1 Unarchiving the menu bar and main window from the main nib file

```
int main (int argc, char* argv[])

{

    IBNibRef        nibRef;

    WindowRef       window;

    OSStatus        err;
```

```
    // Create a nib reference to a nib file.
    err = CreateNibReference (CFSTR ("main"), &nibRef);
    // Call the macro require_noerr to make sure no errors occurred
    require_noerr (err, CantGetNibRef);

    // Unarchive the menu bar and make it ready to use.
    err = SetMenuBarFromNib (nibRef, CFSTR("MainMenu"));
    require_noerr (err, CantSetMenuBar);

    // Unarchive the main window.
    err = CreateWindowFromNib (nibRef, CFSTR("MainWindow"), &window);
    require_noerr (err, CantCreateWindow);

    // Dispose of the nib reference as soon as you don't need it any more.
    DisposeNibReference (nibRef);
    // Make the unarchived window visible.
    ShowWindow (window);

    // Start the event loop. RunApplicationEventLoop is a
     // Carbon Event Manager function.
    RunApplicationEventLoop ();

    // You'll jump to one of the "Cant" statements only if there's
    // an error.
    CantCreateWindow:
    CantSetMenuBar:
    CantGetNibRef:
    return err;
}
```

Unarchiving an Object from an Auxiliary Nib File

For most applications, it is useful to factor your application's interface objects into several nib files. The main nib file should contain at the most, the menu bar and the window that opens (if any) when your application starts up. Document windows, palettes, toolbars, contextual menus, and other interface objects should be stored in separate nib files.

The steps for opening an auxiliary nib file and unarchiving an object from it are similar to those used for your main nib file:

1. Call the function `CreateNibReference` to create a reference to the auxiliary nib file that contains the object you want to unarchive.
2. Call the appropriate function to unarchive the object from the nib file. To unarchive a window, call the function `CreateWindowFromNib`; to unarchive a menu, call the function `CreateMenuFromNib`.

5. After you have unarchived the object from the auxiliary nib file, dispose of the nib reference by calling the function `DisposeNibReference`.

One common use of an auxiliary nib file is to store an object that's used repeatedly in an application, such as a document window. Another use is to store objects that are rarely needed, such as an About window. Listing 5-2 shows how to implement a `MyCreateNewDocument` function that your application would call each time the user creates a new document. The code uses macro-based error checking to abort the operation if an error occurs.

Listing 5-2 Unarchiving a document window from an auxiliary nib file

```
WindowRef MyCreateNewDocument (CFStringRef inName)

{

    IBNibRef documentNib;

    OSStatus err;

    WindowRef theWindow;



    // Create a nib reference to an auxiliary nib file with

    // the name document.nib.

    err = CreateNibReference (CFSTR ("document"), &documentNib);

    // Call the macro require_noerr to make sure no errors occurred

    require_noerr (err, CantGetNibRef);



    // Unarchive the document window. Use the name you gave to the

    // window object in the Instances pane in Interface Builder.

    err = CreateWindowFromNib (documentNib, CFSTR("MyDocument"),

                                            &theWindow);

    require_noerr (err, CantCreateWindow);



    // Dispose of the nib reference as soon as you don't need it anymore.

    DisposeNibReference (documentNib);



    // Call the Window Manager function to set the title shown in the

    // window's title bar to the name passed to MyCreateNewDocument.

    err =  SetWindowTitleWithCFString (theWindow, inName);

    // In this example, the window gets returned. Remember, it's been

    // unarchived, but it is still not visible. It won't be visible

    // until you call the Window Manager function ShowWindow.

    return theWindow;



    // You'll jump to one of the "Cant" statements only if there's

    // an error.
```

```
    CantCreateWindow:

    CantGetNibRef:

     return NULL;

}
```

Unarchiving an Object from an External Bundle

Your application is not limited to using interface objects contained within its own bundle. You can unarchive interface objects from another bundle or framework to which your application has access. For example, you could unarchive a tools palette or other object provided by a plug-in bundle.

The steps for unarchiving an object from a nib file in a framework or other bundle are similar to those used to open an auxiliary nib file and are listed below. The main difference is that you must first create a a reference to the external bundle. You must also call the function `CreateNibReferenceWithCFBundle` instead of `CreateNibReference` to create a reference to the nib file. The steps are as follows:

1. Call the Core Foundation URL Services function `CFURLCreateWithFileSystemPath` to create a URL that points to the desired bundle. (For reference documentation, see *CFURL Reference*.)
2. Call the Core Foundation Bundle Services function `CFBundleCreate` to create a reference to the bundle that contains nib file you want to open. (For reference documentation, see *CFBundle Reference*.)
3. Call the function `CreateNibReferenceWithCFBundle` to create a reference to the nib file that contains the object you want to unarchive.
4. Call the appropriate function to unarchive the object from the nib file. To unarchive a window, call the function `CreateWindowFromNib`; to unarchive a menu, call the function `CreateMenuFromNib`.
5. After you have unarchived the object from the nib file, dispose of the nib reference by calling the function `DisposeNibReference`.

The function `MyCreateWidgetFromFramework`, shown in Listing 5-3, shows how to unarchive a "widget window" from a bundle whose path you pass to the function.

Listing 5-3 Unarchiving a widget window from a nib file in a bundle

```
WindowRef MyCreateWidgetFromBundle (CFStringRef widgetBundlePath

                                    CFStringRef widgetFileName,

                                    CFStringRef  widgetWindowName)

{

    IBNibRef widgetNib;

    OSStatus err;

    WindowRef theWindow;

    CFBundleRef mainBundle;

    CFURLRef    bundleURL;

    CFBundleRef widgetBundle;



    // Look for a resource in the bundle passed to

    // the function MyCreateWidgetFromBundle

    bundleURL = CFURLCreateWithFileSystemPath(

                            kCFAllocatorDefault,

                            widgetBundlePath,

                            kCFURLPOSIXPathStyle,

                            TRUE);

    // Make a bundle instance using the URL Reference

    widgetBundle = CFBundleCreate (kCFAllocatorDefault, bundleURL);
```

```
    // Create a nib reference to the nib file.
    err = CreateNibReferenceWithCFBundle (widgetBundle,
                          widgetFileName, &widgetNib);
    // Call the macro require_noerr to make sure no errors occurred
    require_noerr (err, CantGetNibRef);

    // Unarchive the widget window.
    err = CreateWindowFromNib (widgetNib, widgetWindowName, &theWindow);
    require_noerr (err, CantCreateWindow );

    // Dispose of the nib reference as soon as you don't need it anymore.
    DisposeNibReference (widgetNib);
    // Release the Core Foundation objects
    CFRelease (bundleURL);
    CFRelease (widgetBundle);

    // In this example, the window gets returned. Remember, it's been
    // unarchived, but it is still not visible. It won't be visible
    // until you call the Window Manager function ShowWindow.
    return theWindow;

    // You'll jump to one of the "Cant" statements only if there's
    // an error.
    CantCreateWindow:
    CantGetNibRef:
     return NULL;
}
```

Last updated: 2009-01-06

https://web.archive.org/web/20100115044038/http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html#//apple_ref/doc/uid/10000051i-CH4-SW8

Nib Files

Nib files play an important role in the creation of applications in Mac OS X and iPhone OS. Nib files let you create and manipulate your user interfaces graphically, using the Interface Builder application, instead of programmatically. Because you can see the results of your changes instantly, this gives you the ability to experiment with different layouts and configurations very quickly. It also gives you the flexibility to change many aspects of your user interface later without rewriting any code.

For applications built using the AppKit or UIKit frameworks, nib files take on an extra significance. Both of these frameworks support the use of nib files both for the visual layout of windows, views, and controls and also for the integration of those items with the application's event handling code. Interface Builder works in conjunction with these frameworks, and with Xcode, to help you connect the controls of your user interface to the objects in your project that respond to those controls. This integration significantly reduces the amount of setup that is required after a nib file is loaded and also makes it easy to change the relationships between your code and user interface later.

Because the level of support for nib files is more extensive in Cocoa applications than it is for Carbon applications, the following sections focus on the use of nib files with the AppKit and UIKit frameworks. Although Carbon applications can also use nib files, they do so in a very different way than both AppKit and UIKit, and so the use of nib files in Carbon applications is discussed separately in "Carbon Resources."

Note: Although you can create an Objective-C application without using nib files, doing so is very rare and not recommended. Depending on your application, avoiding the use of nib files can involve overriding large amounts of framework behavior to achieve the same results you would get using a nib file.

Anatomy of a Nib File

Nib files are the documents produced by the Interface Builder application. A nib file describes the visual elements of your application's user interface, including windows, views, controls, and many others. It can also describe non-visual elements, such as the objects in your application that manage your windows and views. Most importantly, a nib file describes these objects exactly as they were configured in Interface Builder. At runtime, these descriptions are used to recreate the objects and their configuration inside your application. When you load a nib file at runtime, you get an exact replica of the objects that were in your Interface Builder document. The nib-loading code instantiates the objects and reestablishes all of the properties and connections that were present between the objects in Interface Builder.

The following sections describe how nib files used with the AppKit and UIKit frameworks are organized, the types of objects found in them, and how you use those objects effectively.

About Your Interface Objects

Interface objects are what you add to an Interface Builder document to implement your user interface. When a nib is loaded at runtime, the interface objects are the objects actually instantiated by the nib-loading code. Most new documents in Interface Builder have at least one interface object by default, typically a window or menu resource, and you add more interface objects to a nib file as part of your interface design. This is the most common type of object in a nib file and is typically why you create nib files in the first place.

Besides representing visual objects, such as windows, views, controls, and menus, interface objects can also represent non-visual objects. In nearly all cases, the non-visual objects you add to a nib file are extra controller objects that your application uses to manage the visual objects. Although you could create these objects in your application, it is often more convenient to add them to a nib file and configure them there. Interface Builder provides a generic object that you use specifically when adding controllers and other non-visual objects to a nib file. It also provides the controller objects that are typically used to manage Cocoa bindings.

About the File's Owner

One of the most important objects in a nib file is the File's Owner object. Unlike interface objects, the File's Owner object is a proxy object that is not created when the nib file is loaded. Instead, you create this object in your code and pass it to the nib-loading code. The reason this object is so important is that it is the main link between your application code and the contents of the nib file. More specifically, it is the controller object that is responsible for the contents of the nib file.

In Interface Builder, you can create connections between the File's Owner and the other interface objects in your nib file. When you load the nib file, the nib-loading code recreates these connections using the replacement object you specify. This allows your object to reference objects in the nib file and receive messages from the interface objects automatically.

About the First Responder

In Interface Builder, the First Responder is a proxy object that represents the first object in your application's dynamically determined responder chain. Because the responder chain of an application cannot be determined at design time, the First Responder proxy acts as a stand-in target for any action messages that need to be directed at the application's responder chain. Menu items commonly target the First Responder proxy. For example, the Minimize menu item in the Window menu hides the frontmost window in an application, not just a specific window, and the Copy menu item should copy the current selection, not just the selection of a single control or view. Other objects in your application can target the First Responder as well.

When you load a nib file into memory, there is nothing you have to do to manage or replace the First Responder proxy object. The AppKit and UIKit frameworks automatically set and maintain the first responder based on the application's current configuration.

For more information about the responder chain and how it is used to dispatch events in AppKit–based applications, see Event Architecture in *Cocoa Event-Handling Guide*. For information about the responder chains and handling actions in iPhone applications, see *iPhone Application Programming Guide*.

About the Top-Level Objects

When your program loads a nib file, Cocoa recreates the entire graph of objects you created in Interface Builder. This object graph includes all of the windows, views, controls, cells, menus, and custom objects found in the nib file. The **top-level objects** are the subset of these objects that do not have a parent object. The top-level objects typically include only the windows, menubars, and custom controller objects that you add to the nib file. (Objects such as File's Owner, First Responder, and Application are proxy objects and not considered top-level objects.) In Interface Builder, you can see the top-level objects in the nib document window when it is set to icon mode (see the circled items in Figure 2-1).

Figure 2-1 Top-level objects in a nib file

Typically, you use outlets in the File's Owner object to store references to the top-level objects of a nib file. If you do not use outlets, however, you can retrieve the top-level objects from the nib-loading routines directly. You should always keep a pointer to these objects somewhere because your application is responsible for releasing them when it is through using them. For more information about the nib object behavior at load time, see "Nib Object Retention."

About Image and Sound Resources

In Interface Builder, you can associate external image and sound resources with the contents of your nib files. Some controls and views are able to display images or play sounds as part of their default configuration. The Interface Builder library provides access to the image and sound resources of your Xcode projects so that you can link your nib files to these resources. The nib file does not store these resources directly. Instead, it stores the name of the resource file so that the nib-loading code can find it later.

When you load a nib file that contains references to image or sound resources, the nib-loading code also reads in those resource files and caches them in memory. In Mac OS X, image and sound resources are stored in named caches so that you can access them later if needed. In iPhone OS, only image resources are stored in named caches. To access images, you use the `imageNamed:` method of `NSImage` or `UIImage`, depending on your platform. To access cached sounds in Mac OS X, use the `soundNamed:` method of `NSSound`.

Nib File Design Guidelines

When creating your nib files, it is important to think carefully about how you intend to use the objects in that file. A very simple application might be able to store all of its user interface components in a single nib file, but for most applications, it is better to distribute components across multiple nib files. Creating smaller nib files lets you load only those portions of your interface that you need immediately. Smaller nib files results in better performance for your application. They also make it easier to debug any problems you might encounter, since there are fewer places to look for problems.

When creating your nib files, try to keep the following guidelines in mind:

- Design your nib files with lazy loading in mind. Plan on loading nib files that contain only those objects you need right away.
- In the main nib file for a Mac OS X application, consider storing only the application menu bar and an optional application delegate object in the nib file. Avoid including any windows or user-interface elements that will not be used until after the application has launched. Instead, place those resources in separate nib files and load them as needed after launch.
- Store repeated user-interface components (such as document windows) in separate nib files.
- For a window or menu that is used only occasionally, store it in a separate nib file. By storing it in a separate nib file, you load the resource into memory only if it is actually used.
- Make the File's Owner the single point-of-contact for anything outside of the nib file; see "Accessing the Contents of a Nib File."

The Nib Object Life Cycle

When a nib file is loaded into memory, the nib-loading code takes several steps to that ensure the objects in the nib file are created and initialized properly. Understanding these steps can help you write better controller code to manage your user interfaces.

The Object Loading Process

When you use the methods of `NSNib` or `NSBundle` to load and instantiate the objects in a nib file, the underlying nib-loading code does the following:

1. It loads the contents of the nib file and any referenced resource files into memory:
    - The raw data for the entire nib object graph is loaded into memory but is not unarchived.
    - Any custom image resources associated with the nib file are loaded and added to the Cocoa image cache; see "About Image and Sound Resources."
    - Any custom sound resources associated with the nib file are loaded and added to the Cocoa sound cache; see "About Image and Sound Resources."
2. It unarchives the nib object graph data and instantiates the objects. How it initializes each new object depends on the type of the object and how it was encoded in the archive by Interface Builder. The nib-loading code uses the following rules (in order) to determine which initialization method to use.
    1. Standard Interface Builder objects (and custom subclasses of those objects) receive an `initWithCoder:` message.

        In Mac OS X, the list of standard objects includes the views, cells, menus, and view controllers that are provided by the system and available in the default Interface Builder library. It also includes any third-party objects that were added to the Interface Builder library using a custom plug-in. Even if you change the class of such an object, Interface Builder encodes the standard object into the nib file and then tells the archiver to swap in your custom class when the object is unarchived.

        In iPhone OS, any object that conforms to the `NSCoding` protocol is initialized using the `initWithCoder:` method. This includes all subclasses of `UIView` and `UIViewController` whether they are part of the default Interface Builder library or custom classes you define.

    2. Custom views in Mac OS X receive an `initWithFrame:` message.

        Custom views are subclasses of `NSView` for which Interface Builder does not have an available implementation. Typically, these are views that you define in your application and use to provide custom visual content. Custom views do not include standard system views (like `NSSlider`) that are part of the default Interface Builder library or part of an integrated third-party plug-in.

        When it encounters a custom view, Interface Builder encodes a special `NSCustomView` object into your nib file. The custom view object includes the information it needs to build the real view subclass you specified. At load time, the `NSCustomView` object sends an `alloc` and `initWithFrame:` message to the real view class and then swaps the resulting view object in for itself. The net effect is that the real view object handles subsequent interactions during the nib-loading process.

        Custom views in iPhone OS do not use the `initWithFrame:` method for initialization.

    3. Custom objects other than those described in the preceding steps receive an `init` message.
3. It reestablishes all connections (actions, outlets, and bindings) between objects in the nib file. This includes connections to File's Owner and other proxy objects. The approach for establishing connections differs depending on the platform:
    - Outlet connections
        - In Mac OS X, the nib-loading code tries to reconnect outlets using the object's own methods first. For each outlet, Cocoa looks for a method of the form `set`*OutletName*`:` and calls it if such a method is present. If it cannot find such a method, Cocoa searches the object for an instance variable with the corresponding

outlet name and tries to set the value directly. If the instance variable cannot be found, no connection is created.

In Mac OS X v10.5 and later, setting an outlet also generates a key-value observing (KVO) notification for any registered observers. These notifications may occur before all inter-object connections are reestablished and definitely occur before any `awakeFromNib` methods of the objects have been called. Prior to v10.5, these notifications are not generated. For more information about KVO notifications, see *Key-Value Observing Programming Guide*.

- In iPhone OS, the nib-loading code uses the `setValue:forKey:` method to reconnect each outlet. That method similarly looks for an appropriate accessor method and falls back on other means when that fails. For more information about how this method sets values, see its description in *NSKeyValueCoding Protocol Reference*.

  Setting an outlet in iPhone OS also generates a KVO notification for any registered observers. These notifications may occur before all inter-object connections are reestablished and definitely occur before any `awakeFromNib` methods of the objects have been called. For more information about KVO notifications, see *Key-Value Observing Programming Guide*.

- Action connections
  - In Mac OS X, the nib-loading code uses the source object's `setTarget:` and `setAction:` methods to establish the connection to the target object. If the target object does not respond to the action method, no connection is created. If the target object is `nil`, the action is handled by the responder chain.
  - In iPhone OS, the nib-loading code uses the `addTarget:action:forControlEvents:` method of the `UIControl` object to configure the action. If the target is `nil`, the action is handled by the responder chain.
- Bindings
  - In Mac OS X, Cocoa uses the `bind:toObject:withKeyPath:options:` method of the source object to create the connection between it and its target object.
  - Bindings are not supported in iPhone OS.

4. It sends an `awakeFromNib` message to the appropriate objects in the nib file that define the matching selector:
   - In Mac OS X, this message is sent to any interface objects that define the method. It is also sent to the File's Owner and any proxy objects that define it as well.
   - In iPhone OS, this message is sent only to the interface objects that were instantiated by the nib-loading code. It is not sent to File's Owner, First Responder, or any other proxy objects.
5. It displays any windows whose "Visible at launch time" attribute was enabled in Interface Builder.

The order in which the nib-loading code calls the `awakeFromNib` methods of objects is not guaranteed. In Mac OS X, Cocoa tries to call the `awakeFromNib` method of File's Owner last but does not guarantee that behavior. If you need to configure the objects in your nib file further at load time, the most appropriate time to do so is after your nib-loading call returns. At that point, all of the objects are created, initialized, and ready for use.

Nib Object Retention

Each time you ask the `NSBundle` or `NSNib` class to load a nib file, the underlying code creates a new copy of the objects in that file and returns them to you. The nib-loading code does not recycle nib file objects from a previous load attempt. Because each set of objects is a new copy, your code is responsible for releasing those objects when it is done with them. How you release the objects depends on the platform and on the memory model in use. Table 2-1 lists the supported platform and memory model configurations and the nib retention behavior associated with each one.

Table 2-1 Object retention rules for nib objects

Configuration	Description
Mac OS X - managed memory model	Objects in the nib file are initially created with a retain count of 1. As it rebuilds the object hierarchy, however, AppKit autoreleases any objects that have a parent or owning object, such as views nested inside view hierarchies. By the time the nib-loading code is done, only the top-level objects in the nib file have a positive retain count and no owning object. Your code is responsible for releasing these top-level objects.
Mac OS X - garbage collected memory model	Most objects in the graph are kept in memory through strong references between the objects. Only the top-level objects in the nib file do not have strong references initially. Thus, your code must create strong references to these objects to prevent the object graph from being released.
iPhone OS - managed memory model	Objects in the nib file are created with a retain count of 1 and then autoreleased. As it rebuilds the object hierarchy, however, UIKit reestablishes connections between the objects using the `setValue:forKey:` method, which uses the available setter method or retains the object by default if no setter method is available. If you define outlets for nib-file objects, you should also define a setter method for accessing that outlet. Setter methods for outlets should retain their values, and setter methods for outlets containing top-level objects must retain their values to prevent them from being deallocated. If you do not store the top-level objects in outlets, you must retain either the array returned by the `loadNibNamed:owner:options:` method or the objects inside the array to prevent those objects from being released prematurely.

For both Mac OS X and UIKit, the recommended way to manage the top-level objects in a nib file is to create outlets for them in the File's Owner object and then define setter methods to retain and release those objects as needed. Setter methods give you an appropriate place to include your memory-management code, even in situations where your application uses garbage collection. One easy way to implement your setter methods is to use the `@property` syntax and let the compiler create them for you. For more information on how to define properties, see *The Objective-C Programming Language*.

Built-In Support For Nib Files

The AppKit and UIKit frameworks both provide a certain amount of automated behavior for loading and managing nib files in an application. Both frameworks provide infrastructure for loading an application's main nib file. In addition, the AppKit framework provides support for loading other nib files through the `NSDocument` and `NSWindowController` classes. The following sections describe the built-in support for nib files, how you can take advantage of it, and ways to modify that support in your own applications.

Loading the Main Nib File

Most of the Xcode project templates for applications come preconfigured with a main nib file already in place. All you have to do is modify this default nib file in Interface Builder and build your application. At launch time, the application's default configuration data tells the application object where to find this nib file so that it can load it. In applications based on either AppKit and UIKit, this configuration data is located in the application's `Info.plist` file. When an application is first loaded, the default application startup code looks in the `Info.plist` file for the `NSMainNibFile` key. If it finds it, it looks in the application bundle for a nib file whose name (with or without the filename extension) matches the value of that key and loads it.

Document and Window Controller Nib Files

In the AppKit framework, the `NSDocument` class works with the default window controller to load the nib file containing your document window. The `windowNibName` method of `NSDocument` is a convenience method that you can use to specify the nib file containing the corresponding document window. When a new document is created, the document object passes the nib file name you specify to the default window controller object, which loads and manages the contents of the nib file. If you use the standard templates provided by Xcode, the only thing you have to do is add the contents of your document window to the nib file.

The `NSWindowController` class also provides automatic support for loading nib files. If you create custom window controllers programmatically, you have the option of initializing them with an `NSWindow` object or with the name of a nib file. If you choose the latter option, the `NSWindowController` class automatically loads the specified nib file the first time a client tries to access the window. After that, the window controller keeps the window around in memory; it does not reload it from the nib file, even if the window's "Release when closed" attribute is set in Interface Builder.

Important: When using either `NSWindowController` or `NSDocument` to load windows automatically, it is important that your nib file be configured correctly. Both classes include a `window` outlet that you must connect to the window you want them to manage. If you do not connect this outlet to a window object, the nib file is loaded but the document or window controller does not display any windows. For more information about the Cocoa document architecture, see *Document-Based Applications Overview*.

Changing the Nib Files from the Xcode Defaults

When you create a new Cocoa application project in Xcode, the project template comes preconfigured with one or more nib files. If you have existing nib files you want to use instead, you can replace the template nib files with your custom nib files. Changing the nib file associated with a document object is easy but changing the application's main nib file is somewhat more involved.

Each document in a Cocoa document-based application has its own nib file for storing the document window and any supporting objects. The `NSDocument` class finds this nib file by calling its own `windowNibName` method, the declaration for which is included in the default document class that comes with the Xcode template. To change the nib file associated with the document, simply change the string returned by the `windowNibName` method of your document class.

The main nib file is the only nib file that is required in a Cocoa application. It is loaded immediately before the application enters its main event loop, and the File's Owner for this nib file is the `NSApplication` object itself. A typical main nib contains only the application menu bar and perhaps an application delegate object to handle any application-related events (such as launch-time notifications). Although many applications include other objects in this nib file, doing so is generally not recommended. Instead, it is always preferable to lazily load other resources only as they are needed. Extra objects consume more memory and require more time to load from disk, both of which can degrade launch-time performance.

In Xcode, every new Cocoa project comes configured with a main nib file, called `MainMenu.xib`, that contains a default menu bar for your project. To change the main nib file for your Cocoa application, do the following:

1. In Interface Builder, open the nib file that you want to make the main nib.
2. Select the File's Owner object of the nib and open the identity inspector.
3. In the identity inspector, set the class of File's Owner to `NSApplication` (or to your custom subclass of `NSApplication` if you define one).
4. Open your application project in Xcode.
5. In the Targets section of the Groups & Files pane, select your application target.
6. Open an inspector (or Info) window for the target and select the Properties tab.
7. In the Main Nib File field, enter the name of your new nib file.

The preceding set of steps also work if you want to change the main nib file for an iPhone application. Instead of the `NSApplication` class, set the class of the File's Owner proxy to `UIApplication`. In addition, you might also want to add a custom object and connect it to the delegate outlet of the `UIApplication`. (You could create the delegate object programmatically from the `UIApplicationMain` function but creating it in the nib file is much more common.)

Loading Nib Files Programmatically

Both Mac OS X and iPhone OS provide convenience methods for loading nib files into your application. Both the AppKit and UIKit framework define additional methods on the NSBundle class that support the loading of nib files. In addition, the AppKit framework also provides the `NSNib` class, which provides similar nib-loading behavior as `NSBundle` but offers some additional advantages that might be useful in specific situations.

As you plan out your application, make sure any nib files you plan to load manually are configured in a way that simplifies the loading process. Choosing an appropriate object for File's Owner and keeping your nib files small can greatly improve their ease of use and memory efficiency. For more tips on configuring your nib files, see "Nib File Design Guidelines."

Loading Nib Files Using NSBundle

The AppKit and UIKit frameworks define additional methods on the `NSBundle` class (using Objective-C categories) to support the loading of nib file resources. The semantics for how you use the methods differs between the two platforms as does the syntax for the methods. In AppKit, there are more options for accessing bundles in general and so there are correspondingly more methods for loading nib files from those bundles. In UIKit, applications can load nib files only from their main bundle and so fewer options are needed. The methods available on the two platforms are as follows:

- AppKit
  - `loadNibNamed:owner:` class method
  - `loadNibFile:externalNameTable:withZone:` class method
  - `loadNibFile:externalNameTable:withZone:` instance method
- UIKit

- `loadNibNamed:owner:options:` instance method

Whenever loading a nib file, you should always specify an object to act as File's Owner of that nib file. The role of the File's Owner is an important one. It is the primary interface between your running code and the new objects that are about to created in memory. All of the nib-loading methods provide a way to specify the File's Owner, either directly or as a parameter in an options dictionary.

One of the semantic differences between the way the AppKit and UIKit frameworks handle nib loading is the way the top-level nib objects are returned to your application. In the AppKit framework, you must explicitly request them using one of the `loadNibFile:externalNameTable:withZone:` methods. In UIKit, the `loadNibNamed:owner:options:` method returns an array of these objects directly. The simplest way to avoid having to worry about the top-level objects in either case is to store them in outlets of your File's Owner object and to make sure the setter methods for those outlets retain their values. Because each platform uses different retain semantics, however, you must be sure to send the proper retain or release messages when appropriate. For information about the retention semantics for nib objects, see "Nib Object Retention."

Listing 2-1 shows a simple example of how to load a nib file using the `NSBundle` class in an AppKit–based application. As soon as the `loadNibNamed:owner:` method returns, you can begin using any outlets that refer to the nib file objects. In other words, the entire nib-loading process occurs within the confines of that single call. The nib-loading methods in the AppKit framework return a Boolean value to indicate whether the load operation was successful.

Listing 2-1 Loading a nib file from the current bundle

```
- (BOOL)loadMyNibFile

{

    // The myNib file must be in the bundle that defines self's class.

    if (![NSBundle loadNibNamed:@"myNib" owner:self])

    {

        NSLog(@"Warning! Could not load myNib file.\n");

        return NO;

    }

    return YES;

}
```

Listing 2-2 shows an example of how to load a nib file in a UIKit–based application. In this case, the method checks the returned array to see if the nib objects were loaded successfully. (Every nib file should have at least one top-level object representing the contents of the nib file.) This example shows the simple case when the nib file contains no proxy objects other than the File's Owner object. For an example of how to specify additional proxy objects, see "Replacing Proxy Objects at Load Time."

Listing 2-2 Loading a nib in an iPhone application

```
- (BOOL)loadMyNibFile

{

    NSArray*    topLevelObjs = nil;



    topLevelObjs = [[NSBundle mainBundle] loadNibNamed:@"myNib" owner:self options:nil];

    if (topLevelObjs == nil)

    {

        NSLog(@"Error! Could not load myNib file.\n");

        return NO;

    }

    return YES;

}
```

Getting a Nib File's Top-Level Objects

The easiest way to get the top-level objects of your nib file is to define outlets in the File's Owner object along with setter methods (or better yet, properties) for accessing those objects. This approach ensures that the top-level objects are retained by your object and that you always have references to them.

Listing 2-3 shows the interface and implementation of a stripped down Cocoa class that uses an outlet to retain the nib file's only top-level object. In this case, the only top-level object in the nib file is an NSWindow object. Because top-level objects in Cocoa have an initial retain count of 1, an extra release message is included. This is fine because by the time the release call is made, the property has already retained the window. You would not want to release top-level objects in this manner in an iPhone application.

Listing 2-3 Using outlets to get the top-level objects

```
// Class interface

@interface MyController : NSObject {

    NSWindow *window;

}

@property(retain) IBOutlet NSWindow *window;

- (void)loadMyWindow;



@end



// Class implementation

@implementation MyController

// The synthesized property retains the window automatically.

@synthesize window;



- (void)loadMyWindow

{

    [NSBundle loadNibNamed:@"myNib" owner:self];



    // The window starts off with a retain count of 1

    // and is then retained by the property, so add an extra release.

    [window release];

}

@end
```

If you do not want to use outlets to store references to your nib file's top-level objects, you must retrieve those objects manually in your code. The technique for obtaining the top-level objects differs depending on the target platform. In Mac OS X, you must ask for the objects explicitly, whereas in iPhone OS they are returned to you automatically.

Listing 2-4 shows the process for getting the top-level objects of a nib file in Mac OS X. This method places a mutable array into the nameTable dictionary and associates it with the NSNibTopLevelObjects key. The nib-loading code looks for this array object and, if present, places the top-level objects in it. Because each object starts with a retain count of 1 before it is added to the array, simply releasing the array is not enough to release the objects in the array as well. As a result, this method sends a release message to each of the objects to ensure that the array is the only entity holding a reference to them.

Listing 2-4 Getting the top-level objects from a nib file at runtime

```
- (NSArray*)loadMyNibFile

{

    NSBundle*           aBundle = [NSBundle mainBundle];

    NSMutableArray*     topLevelObjs = [NSMutableArray array];
```

```
    NSDictionary*       nameTable = [NSDictionary dictionaryWithObjectsAndKeys:
                                        self, NSNibOwner,
                                        topLevelObjs, NSNibTopLevelObjects,
                                        nil];

    if (![aBundle loadNibFile:@"myNib" externalNameTable:nameTable withZone:nil])
    {
        NSLog(@"Warning! Could not load myNib file.\n");
        return nil;
    }

    // Release the objects so that they are just owned by the array.
    [topLevelObjs makeObjectsPerformSelector:@selector(release)];
    return topLevelObjs;
}
```

Obtaining the top-level objects in an iPhone application is much simpler and is shown in Listing 2-2. In the UIKit framework, the `loadNibNamed:owner:options:` method of `NSBundle` automatically returns an array with the top-level objects. In addition, by the time the array is returned, the retain counts on the objects are adjusted so that you do not need to send each object an extra release message. The returned array is the only owner of the objects.

Loading Nib Files Using NSNib

In Mac OS X, the AppKit framework supports the loading of nib files using the `NSNib` class. You can use the `NSNib` class to load nib files that reside outside of a bundle or in situations where you plan to load the same nib file more than once. Loading a nib file with this class is always a two-step process. First, you create an instance of the `NSNib` class, initializing it with the nib file's location information. Second, you instantiate the contents of the nib file to load the objects into memory. Each time you instantiate the nib file, you specify a different File's Owner object and can also receive a new set of top-level objects.

Note: The `NSNib` class is not available in iPhone OS. You should use the `NSBundle` class to load nib files instead.

When you want to load the same nib file multiple times, the two-step process used by `NSNib` offers some advantages over the `NSBundle` methods. An `NSNib` object loads the nib data into memory only once, but each time you call one of its instantiate methods, you receive a unique copy of the nib file objects. If you need to create several copies of a nib file's objects in quick succession, the caching provided by `NSNib` can improve performance.

Listing 2-5 shows one way to load the contents of a nib file using the `NSNib` class. The array returned to you by the `instantiateNibWithOwner:topLevelObjects:` method comes already autoreleased. If you intend to use that array for any period of time, you should make a copy of it.

Listing 2-5 Loading a nib file using NSNib

```
- (NSArray*)loadMyNibFile
{
    NSNib*      aNib = [[NSNib alloc] initWithNibNamed:@"MyPanel" bundle:nil];
    NSArray*    topLevelObjs = nil;

    if (![aNib instantiateNibWithOwner:self topLevelObjects:&topLevelObjs])
    {
        NSLog(@"Warning! Could not load nib file.\n");
        return nil;
    }
```

```
    // Release the raw nib data.
    [aNib release];

    // Release the top-level objects so that they are just owned by the array.
    [topLevelObjs makeObjectsPerformSelector:@selector(release)];

    // Do not autorelease topLevelObjs.
    return topLevelObjs;
}
```

Replacing Proxy Objects at Load Time

In iPhone OS, it is possible to create nib files that include proxy objects besides the File's Owner. Proxy objects represent objects created outside of the nib file but which have some connection to the nib file's contents. Proxies are commonly used to support navigation controllers in iPhone applications. When working with navigation controllers, you typically connect the File's Owner object to some common object such as your application delegate. Proxy objects therefore represent the parts of the navigation controller object hierarchy that are already loaded in memory, either because they were created programmatically or loaded from a different nib file.

Note: Custom proxy objects (other than File's Owner) are not supported in Mac OS X nib files.

Each proxy object you add to a nib file must have a unique name. To assign a name to an object, select the object in Interface Builder and open the inspector window. The Attributes pane of the inspector contains a Name field, which you use to specify the name for your proxy object. The name you assign should be descriptive of the object's behavior or type, but really it can be anything you want.

When you are ready to load a nib file containing proxy objects, you must specify the replacement objects for any proxies when you call the `loadNibNamed:owner:options:` method. The *options* parameter of this method accepts a dictionary of additional information. You use this dictionary to pass in the information about your proxy objects. The dictionary must contain the `UINibProxiedObjectsKey` key whose value is another dictionary containing the name and object for each proxy replacement.

Listing 2-6 shows a sample version of an `applicationDidFinishLaunching:` method that loads the application's main nib file manually. Because the application's delegate object is created by the `UIApplicationMain` function, this method uses a proxy (with the name "AppDelegate") in the main nib file to represent that object. The proxies dictionary stores the proxy object information and the options dictionary wraps that dictionary.

Listing 2-6 Replacing proxy objects in a nib file

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    NSArray*    topLevelObjs = nil;
    NSDictionary*    proxies = [NSDictionary dictionaryWithObject:self forKey:@"AppDelegate"];
    NSDictionary*    options = [NSDictionary dictionaryWithObject:proxies forKey:UINibProxiedObjectsKey];

    topLevelObjs = [[NSBundle mainBundle] loadNibNamed:@"Main" owner:self options:options];
    if ([topLevelObjs count] == 0)
    {
        NSLog(@"Warning! Could not load myNib file.\n");
        return;
    }

    // Show window
    [window makeKeyAndVisible];
}
```

For more information about the options dictionary of the `loadNibNamed:owner:options:` method, see *NSBundle UIKit Additions Reference*.

Accessing the Contents of a Nib File

Upon the successful loading a nib file, its contents become ready for you to use immediately. If you configured outlets in your File's Owner to point to nib file objects, you can now use those outlets. If you did not configure your File's Owner with any outlets, you should make sure you obtain a reference to the top-level objects in some manner so that you can release them later.

Because outlets are populated with real objects when a nib file is loaded, you can subsequently use outlets as you would any other object you created programmatically. For example, if you have an outlet pointing to a window, you could send that window a `makeKeyAndOrderFront:` message to show it on the user's screen. When you are done using the objects in your nib file, you must release them like any other objects.

Important: You are responsible for releasing the top-level objects of any nib files you load when you are finished with those objects. Failure to do so is a cause of memory leaks in many applications. After releasing the top-level objects, it is a good idea to clear any outlets pointing to objects in the nib file by setting them to `nil`. You should clear outlets associated with all of the nib file's objects, not just the top-level objects.

Connecting Menu Items Across Nib Files

The items in a Mac OS X application's menu bar often need to interact with many different objects, including your application's documents and windows. The problem is that many of these objects cannot (or should not) be accessed directly from the main nib file. The File's Owner of the main nib file is always set to an instance of the `NSApplication` class. And although you might be able to instantiate a number of custom objects in your main nib file, doing so is hardly practical or necessary. In the case of document objects, connecting directly to a specific document object is not even possible because the number of document objects can change dynamically and can even be zero.

Most menu items send action messages to one of the following:

- A fixed object that always handles the command
- A dynamic object, such as a document or window

Messaging fixed objects is a relatively straightforward process that is usually best handled through the application delegate. The application delegate object assists the `NSApplication` object in running the application and is one of the few objects that rightfully belongs in the main nib file. If the menu item refers to an application-level command, you can implement that command directly in the application delegate or just have the delegate forward the message to the appropriate object elsewhere in your application.

If you have a menu item that acts on the contents of the frontmost window, you need to link the menu item to the First Responder proxy object. If the action method associated with the menu item is specific to one of your objects (and not defined by Cocoa), you must add that action to the First Responder before creating the connection. To set up a First Responder connection in Interface Builder v3.0 and later, do the following:

1. For custom actions, add the action to the First Responder proxy object:
    1. In your main nib file, select the First Responder proxy object.
    2. Open the inspector window and select the Identity pane.
    3. In the Class Actions section, click the plus (+) button to add the new action method. The new action's name is initially selected.
    4. Enter the name of your action and press Return.
2. Open the menu bar resource.
3. If you have not already done so, add your menu item to the desired menu.
4. Control-click the menu item and drag it to the First Responder proxy object in the nib document window.
5. In the Connections tab of the inspector window, select your action and click Connect.

After creating the connection, you need to implement the action method in your `NSDocument` or `NSResponder` subclass. That object should also implement the `validateMenuItem:` method to enable the menu item at appropriate times. For more information about how the responder chain handles commands, see *Cocoa Event-Handling Guide*.

https://web.archive.org/web/20091122060420/http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Conceptual/LoadingResources/MOSXResources/MOSXResources.html#//apple_ref/doc/uid/10000051i-CH3-SW1

About Resources

There are several reasons to use resources in your application:

- They can reduce the amount of code needed to create your application's user interface.
- They make it possible to change your application's user interface without changing any code.
- They make it easy to localize your application's user-visible content.
- They can store other types of custom data that might be difficult or time-consuming to create at runtime.

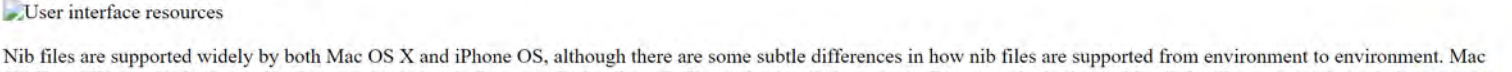
The following sections describe the types of resources typically found in applications and how you use them.

Nib Files

Nib files are the quintessential resource type used to create graphical applications. A nib file is a data archive describing the objects, configuration, and layout information associated with an application's user interface. You create nib files using the Interface Builder application, which provides a graphical assembly area for the windows and menus that comprise applications. You assemble windows and menus in Interface Builder by dragging and dropping custom views, controls, and other components from the provided library of objects. In addition to positioning items inside a window or view, you can configure the properties of those items as well so that they have the custom look and behavior you want for your application. In Cocoa applications, you can also create connections between objects to facilitate the passing of messages between them.

When you load a nib file, the nib-loading code recreates your objects exactly as they were when you designed them, including any custom configuration or connection information. Because the objects are given to you fully configured, it is possible to create complex user interfaces (see Figure 1-1) with very little code.

Figure 1-1 User interface resources



Nib files are supported widely by both Mac OS X and iPhone OS, although there are some subtle differences in how nib files are supported from environment to environment. Mac OS X and iPhone OS both provide the same basic level of support for loading nib files and using their contents. For example, both provide nib-loading support through the `NSBundle` class and automated support for loading the application's main nib file. In Mac OS X, Cocoa provides additional support for loading nib files associated with documents and for using the `NSNib` class to load nibs. The Carbon environment also supports the use of nib files, although the semantics for using them are different. For Carbon applications, nib files are a repository of user interface items that can be loaded one-by-one, instead of all at once.

Note: The term "nib" and the corresponding file extension are an acronym for "NeXT Interface Builder." The Interface Builder application was originally developed at NeXT Computer, whose OPENSTEP operating system was used as the basis for creating Mac OS X.

For general information on how to use nib files in Cocoa-based applications (including iPhone applications), see "Nib Files." For information on how to use nib files in Carbon applications, see "Carbon Resources." For information on how to create nib files, see *Interface Builder User Guide*.

String Resources

Text strings are a prominent part of most user interfaces. Text strings are commonly found in an application's nib files but may also be found in other places as well. For example, if an error occurs, an application might load a string corresponding to that error and display the string in an alert panel. Instead of hard-coding such strings inside source files, which would make localization much more difficult, an application can instead load them from a strings resource file.

A strings resource file (also known as a **strings file** because its file extension is `.strings`) is a human-readable text file (in the UTF-16 encoding) containing a set of string resources for an application. The purpose of strings files is to provide an external repository for an application's localizable text. An application can have any number of strings files and each strings file can contain any number of strings. Each entry in a strings file consists of a key-value pair where both the key and value are themselves strings. The key portion never changes and represents the identifier that your application uses to retrieve the string; however, the value for that key is typically translated to one of the languages your application supports.

Mac OS X provides tools to help you automatically generate strings files for your application. The tools search your code for any usage of specific string-loading routines and use that code to generate strings files for you. For more information about loading string resources and generating strings files, see "String Resources."

Image, Sound, and Multimedia Resources

Mac OS X and iPhone OS make extensive use of image resources (and to a lesser extent sound and multimedia resources) to create a unique visual style for the entire system. Some of these image resources are used to implement the glossy, three-dimensional texture commonly found in system components, such as the Aqua controls. Apple applications make extensive use of high-quality images to create the look and feel typically associated with the underlying system. Developers are similarly encouraged to use high-quality images to create beautiful and easy-to-use interfaces for their applications. The use of images can not only simplify your drawing code but for complex visual elements can improve performance by providing a prerendered version that can be cached and reused.

Because images are such an important part of graphical user interfaces in general, and Mac OS X and iPhone OS in particular, each system provides extensive support for loading and drawing image resource files. Both Mac OS X and iPhone OS provide support for loading and decoding image files saved in a variety of different formats. For resource files, however, the most commonly used formats include PNG, TIFF, PDF, GIF, and JPEG.

Just as you use images, you can use sound and multimedia resources to create a unique presentation style for your application. Although used less frequently than images, sounds can be used to provide feedback or to alert the user to special events. Audio support is provided by the Core Audio family of frameworks and also by custom classes in the AppKit framework. Similarly, you can use movie clips to present video-based content. Mac OS X also provides extensive use for video and multimedia resources through the QuickTime and QuickTime Kit frameworks. In iPhone OS, similar support is provided by the Media Player framework.

For information about how to use image, sound, and video resources in your applications, see "Image, Sound, and Video Resources."

Property Lists

Property list files are a way to store custom configuration data outside of your application code. Mac OS X and iPhone OS use property lists extensively to implement features such as user preferences and information property list files for bundles. You can similarly use property lists to store private (or public) configuration data for your applications.

A property-list file is essentially a set of structured data values. You can create and edit property lists either programmatically or using the Property List Editor application (located in `/Developer/Applications/Utilities`). The structure of custom property-list files is completely up to you. You can use property lists to store string, number, Boolean, date, and raw data values. By default, a property list stores data in a single dictionary structure, but you can assign additional dictionaries and arrays as values to create a more hierarchical data set.

For information about using property lists, see *Property List Programming Guide* and *Property List Programming Topics for Core Foundation*.

Other Resource Files

In addition to the resource types listed in the preceding sections, Table 1-1 lists some additional resource file types you might find in an application bundle.

Table 1-1 Other resource types

Resource Type	Description
AppleScript files	In Mac OS X, AppleScript terminology and suite files contain information about the scriptability of an application. These files can use the file extensions `.sdef`, `.scriptSuite`, or `.scriptTerminology`. Because the actual AppleScript commands used to script an application are visible in user scripts and the Script Editor application, these resources need to be localized. For information on supporting AppleScript, see *AppleScript Overview*.
Help files	In Mac OS X, help content typically consists of a set of HTML files created using a standard text-editing program and registered with the Help Viewer application. (For information on how to register with Help Viewer, see *Apple Help Programming Guide*.) It is also possible to embed PDF files, RTF files, HTML files or other custom documents in your bundle and open them using an external application, such as Preview or Safari. For information on how to open files, see *Launch Services Programming Guide*.
Custom files	Your application can put custom data files or templates inside the appropriate bundle directories.

Legacy Resource Files

In earlier versions of Mac OS, resources were stored in files that used a `.rsrc` extension. These files were capable of storing multiple resources, including images, sounds, user-interface content, configuration data, and many others. Although support for these files is still available in Mac OS X through the Resource Manager routines, their use is deprecated and strongly discouraged. Old-style resource files are usually holdovers from Carbon applications that were ported from Mac OS 9. Even if you are just now porting such an application to Mac OS X, it still makes much more sense to replace your old resources with new ones. Improvements in Mac OS X, particularly in the area of high-resolution graphics, mean that graphics resources found in these files would look out of place in the current system. In addition, most of the other resource types are obsolete or can be easily replaced by property lists and other resource types.

If you have legacy code and need to know how to access old-style resource files, see *Resource Manager Reference*.

---

https://web.archive.org/web/20100114164340/http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/LoadingResources/Introduction/Introduction.html

Introduction

Applied to computer programs, resources are data files that accompany a program's executable code. Resources simplify the code you have to write by moving the creation of complex sets of data or graphical content outside of your code and into more appropriate tools. For example, rather than creating images pixel by pixel using code, it is much more efficient (and practical) to create them in an image editor. To take advantage of a resource, all your code has to do is load it at runtime and use it.

In addition to simplifying your code, resources are also an intimate part of the internationalization process for all applications. Rather than hard-coding strings and other user-visible content in your application, you can place that content in external resource files. Localizing your application then becomes a simple process of creating new versions of each resource file for each supported language. The bundle mechanism used in both Mac OS X and iPhone OS provides a way to organize localized resources and to facilitate the loading of resource files that match the user's preferred language.

This document provides information about the types of resources supported in Mac OS X and iPhone OS and how you use those resources in your code. This document does not focus on the resource-creation process. Most resources are created using either third-party applications or the developer tools provided in the `/Developer/Applications` directory. In addition, although this document refers to the use of resources in applications, the information also applies to other types of bundled executables, including frameworks and plug-ins.

Before reading this document, you should be familiar with the organizational structure imposed by application bundles. Understanding this structure makes it easier to organize and find the resource files your application uses. For information on the structure of bundles, see *Bundle Programming Guide*.

Organization of This Document

This document includes the following chapters:

- "About Resources" provides an introduction to the resource types supported in Mac OS X and iPhone OS.
- "Nib Files" describes the Cocoa-specific support for nib files.
- "Carbon Resources" describes the Carbon-specific support for nib files.
- "String Resources" describes the support for localized string resources in applications.
- "Image, Sound, and Video Resources" describes the support for image, sound, and video resources in applications.

See Also

The following ADC Reference Library documents are conceptually related to *Resource Programming Guide*:

- *Bundle Programming Guide* describes the bundle structure used by applications to store executable code and resources.
- *Internationalization Programming Topics* describes the process of preparing an application (and it's resources) for translation into other languages.
- *Interface Builder User Guide* describes the application used to create nib file resources.
- *Property List Programming Guide* describes the facilities in place for loading property-list resource files into a Cocoa application.
- *Property List Programming Topics for Core Foundation* describes the facilities in place for loading property-list resource files into a C-based application.

Last updated: 2009-01-06

https://web.archive.org/web/20100711125826/http://developer.apple.com/mac/library/DOCUMENTATION/CoreFoundation/Conceptual/CFBundles/DocumentPackages/DocumentPackages.html

Document Packages

If your document file formats are getting too complex to manage because of several disparate types of data, you might consider adopting a package format for your documents. Document packages give the illusion of a single document to users but provide you with flexibility in how you store the document data internally. Especially if you use several different types of standard data formats, such as JPEG, GIF, or XML, document packages make accessing and managing that data much easier.

Defining Your Document Directory Structure

Apple does not prescribe any specific structure for document packages. The contents and organization of the package directory are left to you. You are encouraged, however, to create either a flat directory structure or use the framework bundle structure. which involves placing your files in a top-level `Resources` subdirectory. (For more information about the bundle structure of frameworks, see "Framework Bundles.")

Registering Your Document Type

To register a document as a package, you must modify the document type information in your application's information property list (`Info.plist`) file. The `CFBundleDocumentTypes` key stores information about the document types your application supports. For each document package type, include the `LSTypeIsPackage` key with an appropriate value. The presence of this key tells the Finder and Launch Services to treat directories with the given file extension as a package. See Property List Key Reference in *Runtime Configuration Guidelines* for more information about `Info.plist` keys.

Document packages should always have an extension to identify them—even though that extension may be hidden by the user. The extension allows the Finder to identify your document directory and treat it as a package. You should never associate a document package with a MIME type or 4-byte OS type.

Creating a New Document Package

When it is time for your application to create a new document, it can do so in one of two ways:

- Use an `NSFileWrapper` object to create the document package.
- Create the document package manually.

If you are creating a Cocoa application, the `NSFileWrapper` class is the preferred way to create document packages because it ties in with the existing support in `NSDocument` for reading and writing your document contents. (The `NSFileWrapper` class is also available in iPhone OS 4.0 and later as part of the Foundation framework.) For information on how to use this class, see *NSFileWrapper Class Reference*.

If you are writing a command-line tool or other type of application that does not use the higher-level Objective-C frameworks (such as AppKit or UIKit), you can still create document packages manually. The important thing to remember about creating a document package is that it is just a directory. As long as the type of the document package is registered (as described in "Registering Your Document Type"), all you have to do is create a directory with the appropriate filename extension. (The Finder uses the filename extension as its cue to treat the directory as a package.) You can create the directory (and create any files you want to put inside that directory) using the standard BSD file system routines or using the `NSFileManager` class in the Foundation framework.

Accessing Your Document Contents

There are several ways to access the contents of a document package. Because a document package is a directory, you can access the document's contents using any appropriate file-system routines. If you use a bundle structure for your document package, you can also use the `NSBundle` or `CFBundleRef` routines. Use of a bundle structure is especially appropriate for documents that store multiple localizations.

If your document package uses a flat directory structure or contains a fixed set of content files, you might find using file-system routines faster and easier than using `NSBundle` or `CFBundleRef`. However, if the contents of your document can fluctuate, you should consider using a bundle structure and `NSBundle` or `CFBundleRef` to simplify the dynamic discovery of files inside your document.

If you are creating a Cocoa application, you must also remember to customize the way your `NSDocument` subclass loads the contents of the document package. The traditional technique of using the `readFromData:ofType:error:` and `dataOfType:error:` methods to read and write data are intended for a single file document. To handle a document package, you must use the `readFromFileWrapper:ofType:error:` and `fileWrapperOfType:error:` methods instead. For information about reading and writing document data from your `NSDocument` subclass, see *Document-Based Applications Overview*.

Last updated: 2010-05-25

https://web.archive.org/web/20100516163746/http://developer.apple.com:80/mac/library/documentation/corefoundation/conceptual/CFBundles/AccessingaBundlesContents/AccessingaBundlesContents.html

Download IconCompanion File PDF

- Table of Contents

NextPrevious

Accessing a Bundle's Contents

When writing bundle-based code, you never use string constants to refer to the location of files in your bundle. Instead, you use the `NSBundle` class or `CFBundleRef` opaque type to obtain the path to the file you want. The reason is that the path to the desired file can vary depending on the user's native language and the bundle's supported localizations. By letting the bundle determine the location of the file, you are always assured of loading the correct file.

This chapter shows you how to use the `NSBundle` class and `CFBundleRef` opaque type to locate files and obtain other information about your bundle. Although these types are not directly interchangeable, they provide comparable features. And at least in Objective-C applications, you can use whichever type suits your needs.

Locating and Opening Bundles

Before you can access a bundle's resources, you must first obtain an appropriate `NSBundle` object or `CFBundleRef` opaque type. The following sections outline the different ways you can get a reference to one of these types.

Getting the Main Bundle

The main bundle is the bundle that contains the code and resources for the running application. If you are an application developer, this is the most commonly used bundle. The main bundle is also the easiest to retrieve because it does not require you to provide any information.

To get the main bundle in a Cocoa application, call the `mainBundle` class method of the `NSBundle` class, as shown in Listing 3-1.

Listing 3-1 Getting a reference to the main bundle using Cocoa

```
NSBundle* mainBundle;



// Get the main bundle for the app.

mainBundle = [NSBundle mainBundle];
```


If you are writing a C-based application, you can use the `CFBundleGetMainBundle` function to retrieve the main bundle for your application, as shown in Listing 3-2.

Listing 3-2 Getting a reference to the main bundle using Core Foundation

```
CFBundleRef mainBundle;



// Get the main bundle for the app

mainBundle = CFBundleGetMainBundle();
```

When getting the main bundle, it is still a good idea to make sure the value you get back represents a valid bundle. When retrieving the main bundle from any application, the returned value might be `NULL` in the following situations:

- If a program is not bundled, attempting to get the main bundle might return a `NULL` value. The bundle code may try to create a main bundle to represent your program's contents, but doing so is not possible in all cases.
- If the agent that launched the program did not specify the full path to the program's executable in the `argv` parameters, the main bundle value might be `NULL`. Bundles rely on either the path to the executable being in `argv[0]` or the presence of the executable's path in the `PATH` environment variable. If neither of these is present, the bundle routines might not be able to find the main bundle directory. Programs launched by `xinetd` often experience this problem when `xinetd` changes the current directory to `/`.

Getting Bundles by Path

If you want to access a bundle other than the main bundle, you can create an appropriate bundle object if you know the path to the bundle directory. Creating a bundle by path is useful in situations where you are defining frameworks or other loadable bundles and know in advance where those bundles will be located.

To obtain the bundle at a specific path using Cocoa, call the `bundleWithPath:` class method of the `NSBundle` class. (You can also use the `initWithPath:` instance method to initialize a new bundle object.) This method takes a string parameter representing the full path to the bundle directory. Listing 3-3 shows an example that accesses a bundle in a local directory.

Listing 3-3 Locating a Cocoa bundle using its path

```
NSBundle* myBundle;



// Obtain a reference to a loadable bundle.

myBundle = [NSBundle bundleWithPath:@"/Library/MyBundle.bundle"];
```

To obtain the bundle at a specific path using Core Foundation, call the `CFBundleCreate` function. When specifying the path location in Core Foundation, you must do so using a `CFURLRef` type. Listing 3-4 shows an example that takes the fixed directory from the preceding example, converts it to a URL, and uses that URL to access the bundle.

Listing 3-4 Locating a Core Foundation bundle using its path

```
CFURLRef bundleURL;

CFBundleRef myBundle;



// Make a CFURLRef from the CFString representation of the

// bundle's path.

bundleURL = CFURLCreateWithFileSystemPath(

                kCFAllocatorDefault,

                CFSTR("/Library/MyBundle.bundle"),

                kCFURLPOSIXPathStyle,

                true );



// Make a bundle instance using the URLRef.

myBundle = CFBundleCreate( kCFAllocatorDefault, bundleURL );



// You can release the URL now.

CFRelease( bundleURL );



// Use the bundle...



// Release the bundle when done.

CFRelease( myBundle );
```

Getting Bundles in Known Directories

Even if you do not know the exact path to a bundle, you can still search for it in some known location. For example, an application with a `PlugIns` directory might want to get a list of all the bundles in that directory. Once you have the path to the directory, you can use the appropriate routines to iterate that directory and return any bundles.

The simplest way to find all of the bundles in a specific directory is to use the `CFBundleCreateBundlesFromDirectory` function. This function returns new `CFBundleRef` types for all of the bundles in a given directory. Listing 3-5 shows how you would use this function to retrieve all of the plug-ins in the application's `PlugIns` directory.

Listing 3-5 Obtaining bundle references for a set of plug-ins

```
CFBundleRef mainBundle = CFBundleGetMainBundle();

CFURLRef plugInsURL;

CFArrayRef bundleArray;



// Get the URL to the application's PlugIns directory.

plugInsURL = CFBundleCopyBuiltInPlugInsURL(mainBundle);



// Get the bundle objects for the application's plug-ins
```

```
// Get the bundle objects for the application's plug ins.

bundleArray = CFBundleCreateBundlesFromDirectory( kCFAllocatorDefault,

                    plugInsURL, NULL );



// Release the CF objects when done with them.

CFRelease( plugInsURL );

CFRelease( bundleArray );
```

Getting Bundles by Identifier

Locating bundles using a bundle identifier is an efficient way to locate bundles that were previously loaded into memory. A bundle identifier is the string assigned to the `CFBundleIdentifier` key in the bundle's `Info.plist` file. This string is typically formatted using reverse-DNS notation so as to prevent name space conflicts with developers in other companies. For example, a Finder plug-in from Apple might use the string `com.apple.Finder.MyGetInfoPlugin` as its bundle identifier. Rather than passing a pointer to a bundle object around your code, clients that need a reference to a bundle can simply use the bundle identifier to retrieve it.

To retrieve a bundle using a bundle identifier in Cocoa, call the `bundleWithIdentifier:` class method of the `NSBundle` class, as shown in Listing 3-6.

Listing 3-6 Locating a bundle using its identifier in Cocoa

```
NSBundle* myBundle = [NSBundle bundleWithIdentifier:@"com.apple.myPlugin"];
```

Listing 3-7 shows how to retrieve a bundle using its bundle identifier in Core Foundation.

Listing 3-7 Locating a bundle using its identifier in Core Foundation

```
CFBundleRef requestedBundle;



 // Look for a bundle using its identifier

 requestedBundle = CFBundleGetBundleWithIdentifier(

        CFSTR("com.apple.Finder.MyGetInfoPlugIn") );
```

Remember that you can only use a bundle identifier to locate a bundle that has already been opened. For example, you could use this technique to open the main bundle and bundles for all statically linked frameworks. You could not use this technique to get a reference to a plug-in that had not yet been loaded.

Searching for Related Bundles

If you are writing a Cocoa application, you can obtain a list of bundles related to the application by calling the `allBundles` and `allFrameworks` class methods of `NSBundle`. These methods create an array of `NSBundle` objects corresponding to the bundles or frameworks currently in use by your application. You can use these methods as convenience functions rather than maintain a collection of loaded bundles yourself.

The `bundleForClass:` class method is another way get related bundle information in a Cocoa application. This method returns the bundle in which a particular class is defined. Again, this method is mostly for convenience so that you do not have to retain a pointer to an `NSBundle` object that you may use only occasionally.

Getting References to Bundle Resources

If you have a reference to a bundle object, you can use that object to determine the location of resources inside the bundle. Cocoa and Core Foundation both provide different ways of locating resources inside a bundle. In addition, you should understand how those frameworks look for resource files within your bundle so as to make sure you put files in the right places at build time.

The Bundle Search Pattern

As long as you use an `NSBundle` object or a `CFBundleRef` opaque type to locate resources, your bundle code need never concern itself with how resources are retrieved from a bundle. Both `NSBundle` and `CFBundleRef` retrieve the appropriate language-specific resource automatically based on the available user settings and bundle configuration. However, you still have to put all those language-specific resources into your bundle, so knowing how they are retrieved is important.

The bundle programming interfaces follow a specific search algorithm to locate resources within the bundle. Global resources have the highest priority, followed by region- and language-specific resources. When considering region- and language-specific resources, the algorithm takes into account both the settings for the current user and development region information in the bundle's `Info.plist` file. The following list shows the order in which resources are searched:

1. Global (nonlocalized) resources
2. Region-specific resources (based on the user's region preferences)
3. Language-specific resources (based on the user's language preferences)
4. Development language of the bundle (as specified by the `CFBundleDevelopmentRegion` in the bundle's `Info.plist` file.)

Important: The bundle interfaces consider case when searching for resource files in the bundle directory. This case-sensitive search occurs even on file systems (such as HFS+) that are not case sensitive when it comes to file names.

Because global resources take precedence over language-specific resources, there should *never* be both a global and localized version of a given resource. If a global version of a resource exists, language-specific versions of the same resource are never returned. The reason for this precedence is performance. If localized resources were searched first, the bundle routines might search needlessly in several localized resource directories before discovering the global resource.

Getting the Path to a Resource

In order to locate a resource file in a bundle, you need to know the name of the file, its type, or a combination of the two. Filename extensions are used to identify the type of a file; therefore, it is important that your resource files include the appropriate extensions. If you used custom subdirectories in your bundle to organize resource files, you can speed up the search by providing the name of the subdirectory that contains the desired file.

Even if you do not have a bundle object, you can still search for resources in directories whose paths you know. Both Core Foundation and Cocoa provide API for searching for files using only path-based information. (For example, in Cocoa you can use the `NSFileManager` object to enumerate the contents of directories and test for the existence of files.) However, if you plan to retrieve more than one resource file, it is always faster to use a bundle object. Bundle objects cache search information as they go, so subsequent searches are usually faster.

Using Cocoa to Find Resources

If you have an `NSBundle` object, you can use the following methods to find the location of resources in that bundle:

- `pathForResource:ofType:`
- `pathForResource:ofType:inDirectory:`
- `pathForResource:ofType:inDirectory:forLocalization:`
- `pathsForResourcesOfType:inDirectory:`
- `pathsForResourcesOfType:inDirectory:forLocalization:`

Suppose you have placed an image called `Seagull.jpg` in your application's main bundle. Listing 3-8 shows you how to retrieve the path for this image file from the application's main bundle.

Listing 3-8 Finding a single resource file using `NSBundle`

```
NSBundle* myBundle = [NSBundle mainBundle];

NSString* myImage = [myBundle pathForResource:@"Seagull" ofType:@"jpg"];
```

If you wanted to look for all image resources in your top-level resource directory, instead of looking for just a single resource, you could use the `pathsForResourcesOfType:inDirectory:` method or one of its equivalents, as shown in Listing 3-9

Listing 3-9 Finding multiple resources using `NSBundle`

```
NSBundle* myBundle = [NSBundle mainBundle];

NSArray* myImages = [myBundle pathsForResourcesOfType:@"jpg"

                            inDirectory:nil];
```

Using Core Foundation to Find Resources

If you have a `CFBundleRef` opaque type, you can use the following methods to find the location of resources in that bundle:

- `CFBundleCopyResourceURL`
- `CFBundleCopyResourceURLInDirectory`
- `CFBundleCopyResourceURLsOfType`
- `CFBundleCopyResourceURLsOfTypeInDirectory`
- `CFBundleCopyResourceURLsOfTypeForLocalization`

Suppose you have placed an image called `Seagull.jpg` in your application's main bundle. Listing 3-10 shows you how to search for this image by name and type using the Core Foundation function `CFBundleCopyResourceURL`. In this case, the code looks for the file named "Seagull" with the file type (filename extension) of "jpg" in the bundle's resource directory.

Listing 3-10 Finding a single resource using a `CFBundleRef`

```
    CFURLRef    seagullURL;



    // Look for a resource in the main bundle by name and type.

    seagullURL = CFBundleCopyResourceURL( mainBundle,

                    CFSTR("Seagull"),

                    CFSTR("jpg"),

                    NULL );
```

Suppose that instead of searching for one image file, you wanted to get the names of all image files in a directory called `BirdImages`. You could load all of the JPEGs in the directory using the function `CFBundleCopyResourceURLsOfType`, as shown in Listing 3-11.

Listing 3-11 Finding multiple resources using a `CFBundleRef`

```
    CFArrayRef  birdURLs;



    // Find all of the JPEG images in a given directory.

    birdURLs = CFBundleCopyResourceURLsOfType( mainBundle,
```

```
                    CFSTR("jpg"),

                    CFSTR("BirdImages") );
```

Note: You can search for resources that do not have a filename extension. To get the path to such a resource, specify the complete name of the resource and specify `NULL` for the resource type.

Opening and Using Resource Files

Once you have a reference to a resource file, you can load its contents and use it in your application. The steps you must take to load and use resource files depends on the type of resource, and as such is not covered in this document. For detailed information about loading and using resources, see *Resource Programming Guide*.

Finding Other Files in a Bundle

With a valid bundle object, you can retrieve the path to the top-level bundle directory as well as paths to many of its subdirectories. Using the available interfaces to retrieve directory paths insulates your code from having to know the exact structure of the bundle or its location in the system. It also allows you to use the same code on different platforms. For example, you could use the same code to retrieve resources from an iPhone application or a Mac OS X application, which have different bundle structures.

To get the path to the top-level bundle directory using Cocoa, you use the `bundlePath` method of the corresponding `NSBundle` object. You can also use the `builtInPlugInsPath`, `resourcePath`, `sharedFrameworksPath`, and `sharedSupportPath` methods to obtain the paths for key subdirectories of the bundle. These methods return path information using an `NSString` object, which you can pass directly to most other `NSBundle` methods or convert to an `NSURL` object as needed.

Core Foundation also defines functions for retrieving several different internal bundle directories. To get the path of the bundle itself, you can use the `CFBundleCopyBundleURL` function. You can also use the `CFBundleCopyBuiltInPlugInsURL`, `CFBundleCopyResourcesDirectoryURL`, `CFBundleCopySharedFrameworksURL`, and `CFBundleCopySupportFilesDirectoryURL` functions to obtain the locations of key subdirectories of the bundle. Core Foundation always returns bundle paths as a `CFURLRef` opaque type. You can use this type to extract a `CFStringRef` type that you can then pass to other Core Foundation routines.

Getting the Bundle's Info.plist Data

One file that every bundle should contain is an information property list (`Info.plist`) file. This file is an XML-based text file that contains specific types of key-value pairs. These key-value pairs specify information about the bundle, such as its ID string, version number, development region, type, and other important properties. (See *Runtime Configuration Guidelines* for the list of keys you can include in this file.) Bundles may also include other types of configuration data, mostly organized in XML-based property lists.

The `NSBundle` class provides the `objectForInfoDictionaryKey:` and `infoDictionary` methods for retrieving information from the `Info.plist` file. The `objectForInfoDictionaryKey:` method returns the localized value for a key and is the preferred method to call. The `infoDictionary` method returns an `NSDictionary` with all of the keys from the property list; however, it does not return any localized values for these keys. For more information, see the *NSBundle Class Reference*.

Core Foundation also offers functions for retrieving specific pieces of data from a bundle's information property list file, including the bundle's ID string, version, and development region. You can retrieve the localized value for a key using the `CFBundleGetValueForInfoDictionaryKey` function. You can also retrieve the entire dictionary of non-localized keys using `CFBundleGetInfoDictionary`. For more information about these and related functions, see *CFBundle Reference*.

Note: Because they take localized values into account, `CFBundleGetValueForInfoDictionaryKey` and `objectForInfoDictionaryKey:` are the preferred interfaces for retrieving keys.

Listing 3-12 demonstrates how to retrieve the bundle's version number from the information property list using Core Foundation functions. Though the value in the information property list may be written as a string, for example "2.1.0b7", the value is returned as an unsigned long integer.

Listing 3-12 Obtaining the bundle's version

```
// This is the 'vers' resource style value for 1.0.0

#define kMyBundleVersion1 0x01008000



UInt32  bundleVersion;



// Look for the bundle's version number.

bundleVersion = CFBundleGetVersionNumber( mainBundle );



// Check the bundle version for compatibility with the app.

if (bundleVersion < kMyBundleVersion1)

    return (kErrorFatalBundleTooOld);
```

Listing 3-13 shows you how to retrieve arbitrary values from the information property list using the `CFBundleGetInfoDictionary` function. The resulting information property list is an instance of the standard Core Foundation type `CFDictionaryRef`. For more information about retrieving information from a Core Foundation dictionary, see *CFDictionary Reference*.

Listing 3-13 Retrieving information from a bundle's information property list

```
CFDictionaryRef bundleInfoDict;

CFStringRef     myPropertyString;



// Get an instance of the non-localized keys.

bundleInfoDict = CFBundleGetInfoDictionary( myBundle );
```

```
    // If we succeeded, look for our property.

    if ( bundleInfoDict != NULL ) {

        myPropertyString = CFDictionaryGetValue( bundleInfoDict,

                    CFSTR("MyPropertyKey") );

    }
```

It is also possible to obtain an instance of a bundle's information dictionary without a bundle object. To do this you use either the Core Foundation function `CFBundleCopyInfoDictionaryInDirectory` or the Cocoa `NSDictionary` class. This can be useful for searching the information property lists of a set of bundles without first creating bundle objects.

Loading and Unloading Executable Code

The key to loading code from an external bundle is finding an appropriate entry point into the bundle's executable file. As with other plug-in schemes, this requires some coordination between the application developer and the plug-in developer. You can publish a custom API for bundles to implement or define a formal plug-in interface. In either case, once you have an appropriate bundle or plug-in, you use the `NSBundle` class (or the `CFBundleRef` opaque type) to access the functions or classes implemented by the external code.

Note: Another option for loading Mach-O code directly is to use the `NSModule` loading routines. However, these routines typically require more work to use and are less preferable than the `NSBundle` or `CFBundleRef` interfaces. For more information, see *Mac OS X ABI Mach-O File Format Reference* in Mac OS X Documentation or see the `NSModule` man pages.

For additional information about loading and unloading code, see *Code Loading Programming Topics for Cocoa*.

Loading Functions

If you are working in C, C++, or even in Objective-C, you can publish your interface as a set of C-based symbols, such as function pointers and global variables. Using the Core Foundation functions, you can load references to those symbols from a bundle's executable file.

You can retrieve symbols using any of several functions. To retrieve function pointers, call either `CFBundleGetFunctionPointerForName` or `CFBundleGetFunctionPointersForNames`. To retrieve a pointer to a global variable, call `CFBundleGetDataPointerForName` or `CFBundleGetDataPointersForNames`. For example, suppose a loadable bundle defines the function shown in Listing 3-14.

Listing 3-14 An example function for a loadable bundle

```
    // Add one to the incoming value and return it.

    long addOne(short number)

    {

        return ( (long)number + 1 );

    }
```

Given a `CFBundleRef` opaque type, you would need to search for the desired function before you could use it in your code. Listing 3-15 shows a code fragment that illustrates this process. In this example, the `myBundle` variable is a `CFBundleRef` opaque type pointing to the bundle.

Listing 3-15 Finding a function in a loadable bundle

```
// Function pointer.

AddOneFunctionPtr addOne = NULL;



// Value returned from the loaded function.

long result = 0;



// Get a pointer to the function.

addOne = (void*)CFBundleGetFunctionPointerForName(

            myBundle, CFSTR("addOne") );



    // If the function was found, call it with a test value.

if (addOne)

{
```

```
{

    // This should add 1 to whatever was passed in

    result = addOne ( 23 );

}
```

Loading Objective-C Classes

If you are writing a Cocoa application, you can load the code for an entire class using the methods of `NSBundle`. The `NSBundle` methods for loading a class are for use with Objective-C classes only and cannot be used to load classes written in C++ or other object-oriented languages.

If a loadable bundle defines a principal class, you can load it using the `principalClass` method of `NSBundle`. The `principalClass` method uses the `NSPrincipalClass` key of the bundle's `Info.plist` file to identify and load the desired class. Using the principal class alleviates the need to agree on specific naming conventions for external classes, instead letting you focus on the behavior of those interfaces. For example, you might use an instance of the principal class as a factory for creating other relevant objects.

If you want to load an arbitrary class from a loadable bundle, call the `classNamed:` method of `NSBundle`. This method searches the bundle for a class matching the name you specify. If the class exists in the bundle, the method returns the corresponding `Class` object, which you can then use to create instances of the class.

Listing 3-16 shows you a sample method for loading a bundle's principal class.

Listing 3-16 Loading the principal class of a bundle

```
- (void)loadBundle:(NSString*)bundlePath

{

    Class exampleClass;

    id newInstance;

    NSBundle *bundleToLoad = [NSBundle bundleWithPath:bundlePath];

    if (exampleClass = [bundleToLoad principalClass])

    {

        newInstance = [[exampleClass alloc] init];

        // [newInstance doSomething];

    }

}
```

For more information about the methods of the `NSBundle` class, see *NSBundle Class Reference*.

Unloading Bundles

In Mac OS X 10.5 and later, you can unload the code associated with an `NSBundle` object using the `unload` method. You might use this technique to free up memory in an application by removing plug-ins or other loadable bundles that you no longer need.

If you used Core Foundation to load your bundle, you can use the `CFBundleUnloadExecutable` function to unload it. If your bundle might be unloaded, you need to ensure that string constants are handled correctly by setting an appropriate compiler flag.

When you compile a bundle with a minimum deployment target of Mac OS X 10.2 (or later), the compiler automatically switches to generating strings that are truly constant in response to `CFSTR("...")`. The compiler also generates these constant strings if you compile with the flag `-fconstant-cfstrings`. Constant strings have many benefits and should be used when possible, however if you reference constant strings after the executable containing them is unloaded, the references will be invalid and will cause a crash. This might happen even if the strings have been retained, for example, as a result of being put in data structures, retained directly, and, in some cases, even copied. Rather than trying to make sure all such references are cleaned up at unload time (and some references might be created within the libraries, making them hard to track), it is best to compile unloadable bundles with the flag `-fno-constant-cfstrings`.

NextPrevious

https://web.archive.org/web/20100515101347/http://developer.apple.com/mac/library/documentation/corefoundation/conceptual/CFBundles/BundleTypes/BundleTypes.html

Download IconCompanion File PDF

- Table of Contents

NextPrevious

Bundle Structures

Bundle structures can vary depending on the type of the bundle and the target platform. The following sections describe the bundle structures used most commonly in both Mac OS X and iPhone OS.

Note: Although bundles are one way of packaging executable code, they are not the only way that is supported. UNIX shell scripts and command-line tools do not use the bundle structure, neither do static and dynamic shared libraries.

Application Bundles

Application bundles are one of the most common types of bundle created by developers. The application bundle stores everything that the application requires for successful operation. Although the specific structure of an application bundle depends on the platform for which you are developing, the way you use the bundle is the same on both platforms. This chapter describes the structure of application bundles in both iPhone OS and Mac OS X.

What Files Go Into an Application Bundle?

Table 2-1 summarizes the types of files you are likely to find inside an application bundle. The exact location of these files varies from platform to platform and some resources may not be supported at all. For examples and more detailed information, see the platform-specific bundle sections in this chapter.

Table 2-1 Types of files in an application bundle

File	Description
`Info.plist` file	(Required) The **information property list** file is a structured file that contains configuration information for the application. The system relies on the presence of this file to identify relevant information about your application and any related files.
Executable	(Required) Every application must have an executable file. This file contains the application's main entry point and any code that was statically linked to the application target.
Resource files	Resources are data files that live outside your application's executable file. Resources typically consist of things like images, icons, sounds, nib files, strings files, configuration files, and data files (among others). Resource files can be localized for a particular language or region or shared by all localizations. The placement of resource files in the bundle directory structure depends on whether you are developing an iPhone or Mac OS X application.

Other support files	Mac OS X applications can embed additional high-level resources such as private frameworks, plug-ins, document templates, and other custom data resources that are integral to the application. Although you can include custom data resources in your iPhone application bundles, you cannot include custom frameworks or plug-ins.

Although most of the resources in an application bundle are optional, this may not always be the case. For example, iPhone applications typically require additional image resources for the application's icon and default screen. And although not explicitly required, most Mac OS X applications include a custom icon instead of the default one provided by the system.

Anatomy of an iPhone Application Bundle

The project templates provided by Xcode do most of the work necessary for setting up your iPhone application bundle during development. However, understanding the bundle structure can help you decide where you should place your own custom files. The bundle structure of iPhone applications is geared more toward the needs of a mobile device. It uses a relatively flat structure with few extraneous directories in an effort to save disk space and simplify access to the files.

The iPhone Application Bundle Structure

A typical iPhone application bundle contains the application executable and any resources used by the application (for instance, the application icon, other images, and localized content) in the top-level bundle directory. Listing 2-1 shows the structure of a simple iPhone application called `MyApp`. The only files that are required to be in subdirectories are those that need to be localized; however, you could create additional subdirectories in your own applications to organize resources and other relevant files.

Listing 2-1 Bundle structure of an iPhone application

```
MyApp.app
    MyApp
    Icon.png
    Info.plist
    Default.png
    MainWindow.nib
    Settings.bundle
    Icon-Settings.png
```

```
iTunesArtwork

en.lproj

   MyImage.png

fr.lproj

   MyImage.png
```

Table 2-2 describes the contents of the application shown in Listing 2-1. Although the application itself is for demonstration purposes only, many of the files it contains represent specific files that iPhone OS looks for when scanning an application bundle. Your own bundles would include some or all of these files depending on the features you support.

Table 2-2 Contents of a typical iPhone application bundle

File	Description
`MyApp`	(Required) The executable file containing your application's code. The name of this file is the same as your application name minus the `.app` extension.
`Icon.png`	(Required) The 57 x 57 pixel icon used to represent your application on the device home screen. This icon should not contain any glossy effects. The system adds those effects for you automatically.
`Info.plist`	(Required) This file contains configuration information for the application, such as its bundle ID, version number, and display name. See "The Information Property List File" for further information.
`Default.png`	The 480 x 320 pixel image to display when your application is launched. The system uses this file as a temporary background until your application loads its window and user interface. If your application does not provide this file, a black background is displayed while the application launches.
`MainWindow.nib`	The application's main nib file contains the default interface objects to load at application launch time. Typically, this nib file contains the application's main window object and an instance of the application delegate object. Other interface objects are then either loaded from additional nib files or created programmatically by the application. (The name of the main nib file can be changed by assigning a different value to the `NSMainNibFile` key in the `Info.plist` file. See "The Information Property List File" for further information.)
`Settings.bundle`	The Settings bundle is a special type of plug-in that contains any application-specific preferences that you want to add to the Settings application. This bundle contains property lists and other resource files to configure and display your preferences.
`Icon-Settings.png`	The 29 x 29 pixel icon used to represent your application in the Settings application. If your application includes a settings bundle, this icon is displayed next to your application name in the Settings application. If you do not specify this icon file, the `Icon.png` file is scaled and used instead.
`iTunesArtwork`	The 512 x 512 icon for an application that is distributed using ad-hoc distribution. This icon would normally be provided by the App Store; because applications distributed in an ad-hoc manner do not go through the App Store, however, it must be present in the application bundle instead. iTunes uses this icon to represent your application. (The file you specify for this property should be the same one you would have submitted to the App Store (typically a JPEG or PNG file), were you to distribute your application that way. The filename must be the one shown at left and must not include a filename extension.)
Custom resource files	Nonlocalized resources are placed at the top level directory and localized resources are placed in language-specific subdirectories of the application bundle. Resources consist of nib files, images, sound files, configuration files, strings files, and any other custom data files you need for your application. For more information about resources, see "Resources in an iPhone Application."

An iPhone application should be internationalized and have a *language*`.lproj` folder for each language it supports. In addition to providing localized versions of your application's custom resources, you can also localize your application icon (`Icon.png`), default image (`Default.png`), and Settings icon (`Icon-Settings.png`) by placing files with the same name in your language-specific project directories. Even if you provide localized versions, however, you should always include a default version of these files at the top-level of your application bundle. The default version is used in situations where a specific localization is not available. For more information about localized resources, see "Localized Resources in Bundles."

The Information Property List File

Every iPhone application must have an information property list (`Info.plist`) file containing the application's confirmation information. When you create a new iPhone application project, Xcode creates this file automatically and sets the value of some of the key properties for you. Table 2-3 lists some additional keys that you should set explicitly. (Xcode obscures actual key names by default, so the string displayed by Xcode is also listed in parenthesis where one is used. You can see the real key names for all keys by Control-clicking the Information Property List key in the editor and choosing Show Raw Keys/Values from the contextual menu that appears.)

Table 2-3 Required keys for the `Info.plist` file

Key	Value
`CFBundleDisplayName` (Bundle display name)	The bundle display name is the name displayed underneath the application icon. This value should be localized for all supported languages.
`CFBundleIdentifier` (Bundle identifier)	The bundle identifier string identifies your application to the system. This string must be a uniform type identifier (UTI) that contains only alphanumeric (`A-Z,a-z,0-9`), hyphen (`-`), and period (`.`) characters. The string should also be in reverse-DNS format. For example, if your company's domain is `Ajax.com` and you create an application named Hello, you could assign the string `com.Ajax.Hello` as your application's bundle identifier. The bundle identifier is used in validating the application signature.
`CFBundleVersion` (Bundle version)	The bundle version string specifies the build version number of the bundle. This value is a monotonically increased string, comprised of one or more period-separated integers. This value cannot be localized.
`LSRequiresIPhoneOS` (Application requires iPhone environment)	A Boolean value that indicates whether the bundle can run on iPhone OS only. Xcode adds this key automatically and sets its value to true. You should not change the value of this key.

In addition to the keys in the preceding table, Table 2-4 lists some keys that are commonly used by iPhone applications. Although these keys are not required, most provide a way to adjust the configuration of your application at launch time. Providing these keys can help ensure that your application is presented appropriately by the system.

Table 2-4 Keys commonly included in the `Info.plist` file

Para	Para
`NSMainNibFile` (Main nib file base name)	A string that identifies the name of the application's main nib file. If you want to use a nib file other than the default one created for your project, associate the name of that nib file with this key. The name of the nib file should not include the `.nib` filename extension.
`UIStatusBarStyle`	A string that identifies the style of the status bar as the application launches. This value is based on the `UIStatusBarStyle` constants declared in `UIApplication.h` header file. The default style is `UIStatusBarStyleDefault`. The application can change this initial status-bar style when it finishes launching. If you do not specify this key, iPhone OS displays the default status bar.
`UIStatusBarHidden`	A Boolean value that determines whether the status bar is initially hidden when the application launches. Set it to true to hide the status bar. The default value is `false`.
`UIInterfaceOrientation`	A string that identifies the initial orientation of the application's user interface. This value is based on the `UIInterfaceOrientation` constants declared in the `UIApplication.h` header file. The default style is `UIInterfaceOrientationPortrait`.
`UIPrerenderedIcon`	A Boolean value that indicates whether the application icon already includes gloss and bevel effects. The default value is `false`. Set it to `true` if you do not want the system to add these effects to your artwork.
`UIRequiredDeviceCapabilities`	An informational key that lets iTunes and the App Store know which device-related features an application requires in order to run. iTunes and the mobile App Store use this list to prevent customers from installing applications on a device that does not support the listed capabilities. The value of this key is either an array or a dictionary. If you use an array, the presence of a given key indicates the corresponding feature is required. If you use a dictionary, you must specify a Boolean value for each key indicating whether the feature is required. In both cases, not including a key indicates that the feature is not required. For a list of keys to include in the dictionary, see *iPhone Application Programming Guide*. This key is supported in iPhone OS 3.0 and later.
`UIRequiresPersistentWiFi`	A Boolean value that notifies the system that the application uses the Wi-Fi network for communication. Applications that use Wi-Fi for any period of time must set this key to `true`; otherwise, after 30 minutes, the device shuts down Wi-Fi connections to save power. Setting this flag also lets the system know that it should display the network selection dialog when Wi-Fi is available but not currently being used. The default value is `false`. Even if the value of this property is `true`, this key has no effect when the device is idle (that is, screen-locked). During that time, the application is considered inactive and, although it may function on some levels, it has no Wi-Fi connection.

Application Icon and Launch Images

The file for the icon displayed in the user's Home screen has the default name of `Icon.png` (although the `CFBundleIconFile` property in the `Info.plist` file lets you rename it). It should be a PNG image file located in the top level of the application bundle. The application icon should be a 57 x 57 pixel image without any shine or round beveling effects. Typically, the system applies these effects to the icon before displaying it. You can override that behavior, however, by including the `UIPrerenderedIcon` key in your application's `Info.plist` file.

Note: If you are distributing your application to local users using ad-hoc distribution (instead of going through the App Store), your bundle should also include a 512 x 512 pixel version of your application icon in a file called `iTunesArtwork`. This file provides the icon that iTunes displays when distributing your application.

The file for the application's launch image is named `Default.png`. This image should closely resemble the application's initial user interface; the system displays the launch image before an application is ready to display its user interface, giving users the impression of a quick launch. The launch image should also be a PNG image file, located in the top level of the application bundle. If the application is launched through a URL, the system looks for a launch image named `Default-`*scheme*`.png`, where *scheme* is the scheme of the URL. If that file is not present, it chooses `Default.png` instead.

To add an image file to a project in Xcode, choose Add to Project from the Project menu, locate the file in the browser, and click Add.

Note: In addition to the icons and launch image at the top level of your bundle, you can also include localized versions of those images in your application's language-specific project subdirectories. For more information about including localized resources in your application, see "Localized Resources in Bundles."

Resources in an iPhone Application

In an iPhone application, nonlocalized resources are located at the top-level of the bundle directory, along with the application's executable file and the `Info.plist` file. Most iPhone applications have at least a few files at this level, including the application's icon, launch image, and one or more nib files. Although you should place most nonlocalized resources in this top-level directory, you can also create subdirectories to organize your resource files. Localized resources must be placed in one or more language-specific subdirectories, which are discussed in more detail in "Localized Resources in Bundles."

Listing 2-2 shows a fictional application that includes both localized and nonlocalized resources. The nonlocalized resources include `Hand.png`, `MainWindow.nib`, `MyAppViewController.nib`, and the contents of the `WaterSounds` directory. The localized resources include everything in the `en.lproj` and `jp.lproj` directories.

Listing 2-2 An iPhone application with localized and nonlocalized resources

```
MyApp.app/
    Info.plist
    MyApp
    Default.png
    Icon.png
    Hand.png
    MainWindow.nib
```

```
    MyAppViewController.nib
    WaterSounds/
        Water1.aiff
        Water2.aiff
    en.lproj/
        CustomView.nib
        bird.png
        Bye.txt
        Localizable.strings
    jp.lproj/
        CustomView.nib
        bird.png
        Bye.txt
        Localizable.strings
```

For information about finding resource files in your application bundle, see "Accessing a Bundle's Contents." For information about how to load resource files and use them in your program, see *Resource Programming Guide*.

Anatomy of a Mac OS X Application Bundle

The project templates provided by Xcode do most of the work necessary for setting up your Mac OS X application bundle during development. However, understanding the bundle structure can help you decide where you should place your own custom files. Mac OS X bundles use a highly organized structure to make it easier for the bundle-loading code to find resources and other important files in the bundle. The hierarchical nature also helps the system distinguish code bundles such as applications from the directory packages used by other applications to implement document types.

The Structure of a Mac OS X Application Bundle

The basic structure of a Mac OS X application bundle is very simple. At the top-level of the bundle is a directory named `Contents`. This directory contains everything, including the resources, executable code, private frameworks, private plug-ins, and support files needed by the application. While the `Contents` directory might seem superfluous, it identifies the bundle as a modern-style bundle and separates it from document and legacy bundle types found in earlier versions of Mac OS.

Listing 2-3 shows the high-level structure of a typical application bundle, including the immediate files and directories you are most likely to find inside the `Contents` directory. This structure represents the core of every Mac OS X application.

Listing 2-3 The basic structure of a Mac OS X application

```
MyApp.app/
    Contents/
        Info.plist
        MacOS/
        Resources/
```

Table 2-5 lists some of the directories that you might find inside the `Contents` directory, along with the purpose of each one. This list is not exhaustive but merely represents the directories in common usage.

Table 2-5 Subdirectories of the `Contents` directory

Directory	Description
`MacOS`	(Required) Contains the application's standalone executable code. Typically, this directory contains only one binary file with your application's main entry point and statically linked code. However, you may put other standalone executables (such as command-line tools) in this directory as well.
`Resources`	Contains all of the application's resource files. This contents of this directory are further organized to distinguish between localized and nonlocalized resources. For more information about the structure of this directory, see "The Resources Directory"
`Frameworks`	Contains any private shared libraries and frameworks used by the executable. The frameworks in this directory are revision-locked to the application and cannot be superseded by any other, even newer, versions that may be available to the operating system. In other words, the frameworks included in this directory take precedence over any other similarly named frameworks found in other parts of the operating system. For information on how to add private frameworks to your application bundle, see *Framework Programming Guide*.

`PlugIns`	Contains loadable bundles that extend the basic features of your application. You use this directory to include code modules that must be loaded into your application's process space in order to be used. You would not use this directory to store standalone executables.
`SharedSupport`	Contains additional non-critical resources that do not impact the ability of the application to run. You might use this directory to include things like document templates, clip art, and tutorials that your application expects to be present but that do not affect the ability of your application to run.

Application bundles have evolved significantly over the years but the overall goal has been the same. The bundle organization makes it easier for the application to find its resources while making it harder for users to interfere with those resources. Because the Finder treats most bundles as opaque entities, it is difficult for casual users to move or delete the resources an application might need.

The Information Property List File

For the Finder to recognize an application bundle as such, you need to include an information property list (`Info.plist`) file. This file contains XML property-list data that identifies the configuration of your bundle. For a minimal bundle, this file would contain very little information, most likely just the name and identifier of the bundle. For more complex bundles, the `Info.plist` file includes much more information.

Important: Bundle resources are located using a case-sensitive search. Therefore, the name of your information property list file must start with a capital "I".

Table 2-6 lists the keys that you should always include in your `Info.plist` file. Xcode provides all of these keys automatically when you create a new project. (Xcode obscures actual key names by default, so the string displayed by Xcode is also listed in parenthesis. You can see the real key names for all keys by Control-clicking the Information Property List key in the editor and choosing Show Raw Keys/Values from the contextual menu that appears.)

Table 2-6 Expected keys in the `Info.plist` file

Key	Description
`CFBundleName` (Bundle name)	The short name for the bundle. The value for this key is usually the name of your application. Xcode sets the value of this key by default when you create a new project.
`CFBundleDisplayName` (Bundle display name)	The localized version of your application name. You typically include a localized value for this key in an `InfoPlist.strings` files in each of your language-specific resource directories.
`CFBundleIdentifier` (Bundle identifier)	The string that identifies your application to the system. This string must be a uniform type identifier (UTI) that contains only alphanumeric (`A-Z`,`a-z`,`0-9`), hyphen (`-`), and period (`.`) characters. The string should also be in reverse-DNS format. For example, if your company's domain is `Ajax.com` and you create an application named Hello, you could assign the string `com.Ajax.Hello` as your application's bundle identifier. The bundle identifier is used in validating the application signature.
`CFBundleVersion` (Bundle version)	The string that specifies the build version number of the bundle. This value is a monotonically increased string, comprised of one or more period-separated integers. This value can correspond to either released or unreleased versions of the application. This value cannot be localized.
`CFBundlePackageType` (Bundle OS Type code)	The type of bundle this is. For applications, the value of this key is always the four-character string `APPL`.
`CFBundleSignature` (Bundle creator OS Type code)	The creator code for the bundle. This is a four-character string that is specific to the bundle. For example, the signature for the TextEdit application is `ttxt`.
`CFBundleExecutable` (Executable file)	The name of the main executable file. This is the code that is executed when the user launches your application. Xcode typically sets the value of this key automatically at build time.

Table 2-7 lists the keys that you should also consider including in your `Info.plist` file.

Table 2-7 Recommended keys for the `Info.plist` file

Key	Description
`CFBundleDocumentTypes` (Document types)	The document types supported by the application. This type consists of an array of dictionaries, each of which provides information about a specific document type.
`CFBundleShortVersionString` (Bundle versions string, short)	The release version of the application. The value of this key is a string comprised of three period-separated integers.
`LSMinimumSystemVersion` (Minimum system version)	The minimum version of Mac OS X required for this application to run. The value for this key is a string of the form *n.n.n* where each *n* is a number representing either the major or minor version number of Mac OS X that is required. For example, the value 10.1.5 would represent Mac OS X v10.1.5.
`NSHumanReadableCopyright` (Copyright (human-readable))	The copyright notice for the application. This is a human readable string and can be localized by including the key in an `InfoPlist.strings` file in your language-specific project directories.
`NSMainNibFile` (Main nib file base name)	The nib file to load when the application is launched (without the `.nib` filename extension). The main nib file is an Interface Builder archive containing the objects (main window, application delegate, and so on) needed at launch time.
`NSPrincipalClass` (Principal class)	The entry point for dynamically loaded Objective-C code. For an application bundle, this is almost always the `NSApplication` class or a custom subclass.

The exact information you put into your `Info.plist` file is dependent on your bundle's needs and can be localized as necessary. For more information on this file, see *Runtime Configuration Guidelines*.

The Resources Directory

The `Resources` directory is where you put all of your images, sounds, nib files, string resources, icon files, data files, and configuration files among others. The contents of this directory are further subdivided into areas where you can store localized and nonlocalized resource files. Nonlocalized resources reside at the top level of the `Resources` directory itself or in a custom subdirectory that you define. Localized resources reside in separate subdirectories called language-specific project directories, which are named to coincide with the specific localization.

The best way to see how the `Resources` directory is organized is to look at an example. Listing 2-4 shows a fictional application that includes both localized and nonlocalized resources. The nonlocalized resources include `Hand.tiff`, `MyApp.icns` and the contents of the `WaterSounds` directory. The localized resources include everything in the `en.lproj` and `jp.lproj` directories or their subdirectories.

Listing 2-4 A Mac OS X application with localized and nonlocalized resources

```
MyApp.app/
    Contents/
        Info.plist
        MacOS/
            MyApp
        Resources/
            Hand.tiff
            MyApp.icns
            WaterSounds/
                Water1.aiff
                Water2.aiff
            en.lproj/
                MyApp.nib
                bird.tiff
                Bye.txt
                InfoPlist.strings
                Localizable.strings
                CitySounds/
                    city1.aiff
                    city2.aiff
            jp.lproj/
                MyApp.nib
                bird.tiff
                Bye.txt
                InfoPlist.strings
                Localizable.strings
                CitySounds/
                    city1.aiff
                    city2.aiff
```

Each of your language-specific project directories should contain a copy of the same set of resource files, and the name for any single resource file must be the same across all localizations. In other words, only the content for a given file should change from one localization to another. When you request a resource file in your code, you specify only the name of the file you want. The bundle-loading code uses the current language preferences of the user to decide which directories to search for the file you requested.

For information about finding resource files in your application bundle, see "Accessing a Bundle's Contents." For information about how to load resource files and use them in your program, see *Resource Programming Guide*.

The Application Icon File

One special resource that belongs in your top-level `Resources` directory is your application icon file. By convention, this file takes the name of the bundle and an extension of `.icns`; the image format can be any supported type, but if no extension is specified, the system assumes `.icns`.

Localizing the Information Property List

Because some of the keys in an application's `Info.plist` file contain user-visible strings, Mac OS X provides a mechanism for specifying localized versions of those strings. Inside each language-specific project directory, you can include an `InfoPlist.strings` file that specifies the appropriate localizations. This file is a strings file (not a property list) whose entries consist of the `Info.plist` key you want to localize and the appropriate translation. For example, in the TextEdit application, the German localization of this file contains the following strings:

```
CFBundleDisplayName = "TextEdit";

NSHumanReadableCopyright = "Copyright © 1995-2009 Apple Inc.\nAlle Rechte vorbehalten.";
```

Creating an Application Bundle

The simplest way to create an application bundle is using Xcode. All new application projects include an appropriately configured application **target**, which defines the rules needed to build an application bundle, including which source files to compile, which resource files to copy to the bundle, and so on. New projects also include a preconfigured `Info.plist` file and typically several other files to help you get started quickly. You can add any custom files as needed using the project window and configure those files using the Info or Inspector windows. For example, you might use the Info window to specify custom locations for resource files inside your bundle.

For information on how to configure targets in Xcode, see *Xcode Build System Guide*.

Framework Bundles

A framework is a hierarchical directory that encapsulates a dynamic shared library and the resource files needed to support that library. Frameworks provide some advantages over the typical dynamic shared library in that they provide a single location for all of the framework's related resources. For example, most frameworks include the header files that define the symbols exported by the framework. Grouping these files with the shared library and its resources makes it easier to install and uninstall the framework and to locate the framework's resources.

Just like a dynamic shared library, frameworks provide a way to factor out commonly used code into a central location that can be shared by multiple applications. Only one copy of a framework's code and resources reside in-memory at any given time, regardless of how many processes are using those resources. Applications that link against the framework then share the memory containing the framework. This behavior reduces the memory footprint of the system and helps improve performance.

Note: Only the code and read-only resources of a framework are shared. If a framework defines writable variables, each application gets its own copy of those variables to prevent it from affecting other applications.

Although you can create frameworks of your own, most developers' experience with frameworks comes from including them in their projects. Frameworks are how Mac OS X delivers many key features to your application. The publicly available frameworks provided by Mac OS X are located in the `/System/Library/Frameworks` directory. In iPhone OS, the public frameworks are located in the `System/Library/Frameworks` directory of the appropriate iPhone SDK directory. For information about adding frameworks to your Xcode projects, see *Xcode Build System Guide*.

Note: The creation of custom frameworks is not supported in iPhone OS.

For more detailed information about frameworks and framework bundles, see *Framework Programming Guide*.

Anatomy of a Framework Bundle

The structure of framework bundles differs from that used by applications and plug-ins. The structure for frameworks is based on a bundle format that predates Mac OS X and supports the inclusion of multiple versions of the framework's code and resources in the framework bundle. This type of bundle is known as a **versioned bundle**. Supporting multiple versions of a framework allows older applications to continue running even as the framework shared library continues to evolve. The bundle's `Versions` subdirectory contains the individual framework revisions while symbolic links at the top of the bundle directory point to the latest revision.

The system identifies a framework bundle by the `.framework` extension on its directory name. The system also uses the `Info.plist` file inside the framework's `Resources` directory to gather information about the configuration of the framework. Listing 2-5 shows the basic structure of a framework bundle. The arrows (`->`) in the listing indicate symbolic links to specific files and subdirectories. These symbolic links provide convenient access to the latest version of the framework.

Listing 2-5 A simple framework bundle

```
MyFramework.framework/

   MyFramework   -> Versions/Current/MyFramework

   Resources     -> Versions/Current/Resources

   Versions/

      A/

         MyFramework

         Headers/

            MyHeader.h

         Resources/

            English.lproj/

               InfoPlist.strings

            Info.plist

      Current   -> A
```

Frameworks are not required to include a `Headers` directory but doing so allows you to include the header files that define the framework's exported symbols. Frameworks can store other resource files in both standard and custom directories.

Creating a Framework Bundle

If you are developing software for Mac OS X, you can create your own custom frameworks and use them privately or make them available for other applications to use. You can create a new framework using a separate Xcode project or by adding a framework target to an existing project.

For information about how to create a framework, see *Framework Programming Guide*.

Loadable Bundles

Plug-ins and other types of loadable bundles provide a way for you to extend the behavior of an application dynamically. A loadable bundle consists of executable code and any resources needed to support that code stored in a bundle directory. You can use loadable bundles to load code lazily into your application or to allow other developers to extend the basic behavior of your application.

Note: The creation and use of loadable bundles is not supported in iPhone OS.

Anatomy of a Loadable Bundle

Loadable bundles are based on the same structure as application bundles. At the top-level of the bundle is a single `Contents` directory. Inside this directory are several subdirectories for storing executable code and resources. The `Contents` directory also contains the bundle's `Info.plist` file with information about the bundle's configuration.

Unlike the executable of an application, loadable bundles generally do not have a `main` function as their main entry point. Instead, the application that loads the bundle is responsible for defining the expected entry point. For example, a bundle could be expected to define a function with a specific name or it could be expected to include information in its `Info.plist` file identifying a specific function or class to use. This choice is left to the application developer who defines the format of the loadable bundle.

Listing 2-6 shows the layout of a loadable bundle. The top-level directory of a loadable bundle can have any extension, but common extensions include `.bundle` and `.plugin`. Mac OS X always treats bundles with those extensions as packages, hiding their contents by default.

Listing 2-6 A simple loadable bundle

```
MyLoadableBundle.bundle

   Contents/

      Info.plist

      MacOS/

         MyLoadableBundle

      Resources/

         Lizard.jpg

         MyLoadableBundle.icns

         en.lproj/

            MyLoadableBundle.nib

            InfoPlist.strings

         jp.lproj/

            MyLoadableBundle.nib

            InfoPlist.strings
```

In addition to the `MacOS` and `Resources` directories, loadable bundles may contain additional directories such as `Frameworks`, `PlugIns`, `SharedFrameworks`, and `SharedSupport`—all the features supported by full-fledged application packages.

The basic structure of a loadable bundle is the same regardless of which language that bundle uses in its implementation. For more information about the structure of loadable bundles, see *Code Loading Programming Topics for Cocoa*.

Creating a Loadable Bundle

If you are developing software for Mac OS X, you can create your own custom loadable bundles and incorporate them into your applications. If other applications export a plug-in API, you can also develop bundles targeted at those APIs. Xcode includes template projects for implementing bundles using either C or Objective-C, depending on the intended target application.

For more information about how to design loadable bundles using Objective-C, see *Code Loading Programming Topics for Cocoa*. For information about how to design loadable bundles using the C language, see *Plug-ins*.

Localized Resources in Bundles

Within the `Resources` directory of a Mac OS X bundle (or the top-level directory of an iPhone application bundle), you can create one or more language-specific project subdirectories to store language- and region-specific resources. The name of each directory is based on the language and region of the desired localization followed by the `.lproj` extension. To specify the language and region, you use the following format:

- *language_region*`.lproj`

The *language* portion of the directory name is a two-letter code that conforms to the ISO 639 conventions. The *region* portion is also a two-letter code but it conforms to the ISO 3166 conventions for designating specific regions. Although the region portion of the directory name is entirely optional, it can be a useful way to tune your localizations for specific parts of the world. For example, you could use a single `en.lproj` directory to support all English speaking nations. However, providing separate localizations for Great Britain (`en_GB.lproj`), Australia (`en_AU.lproj`), and the United States (`en_US.lproj`) lets you tailor your content for each of those countries.

Note: For backwards compatibility, the `NSBundle` class and `CFBundleRef` functions also support human-readable directory names for several common languages, including `English.lproj`, `German.lproj`, `Japanese.lproj`, and others. Although the human-readable names are supported, the ISO names are preferred.

If most of your resource files are the same for all regions of a given language, you can combine a language-only resource directory with one or more region-specific directories. Providing both types of

directories alleviates the need to duplicate every resource file for each region you support. Instead, you can customize only the subset of files that are needed for a particular region. When looking for resources in your bundle, the bundle-loading code looks first in any region-specific directories, followed by the language-specific directory. And if neither localized directory contains the resource, the bundle-loading code looks for an appropriate nonlocalized resource.

Listing 2-7 shows the potential structure of a Mac OS X application that contains both language- and region-specific resource files. (In an iPhone application, the contents of the `Resources` directory would be at the top-level of the bundle directory.) Notice that the region-specific directories contain only a subset of the files in the `en.lproj` directory. If a region-specific version of a resource is not found, the bundle looks in the language-specific directory (in this case `en.lproj`) for the resource. The language-specific directory should always contain a complete copy of any language-specific resource files.

Listing 2-7 A bundle with localized resources

```
Resources/
   MyApp.icns
   en_GB.lproj/
      MyApp.nib
      bird.tiff
      Localizable.strings
   en_US.lproj/
      MyApp.nib
      Localizable.strings
   en.lproj/
      MyApp.nib
      bird.tiff
      Bye.txt
      house.jpg
      InfoPlist.strings
      Localizable.strings
      CitySounds/
         city1.aiff
         city2.aiff
```

For more information on language codes and the process for localizing resources, see *Internationalization Programming Topics*.

NextPrevious

https://web.archive.org/web/20100518063556/http://developer.apple.com/mac/library/documentation/corefoundation/conceptual/CFBundles/AboutBundles/AboutBundles.html

Download IconCompanion File PDF

NextPrevious

About Bundles

Bundles are a convenient way to deliver software in Mac OS X and iPhone OS. Bundles provide a simplified interface for end users and at the same time provide support for development. This chapter provides an introduction to bundles and discusses the role they play in Mac OS X and iPhone OS.

Bundles and Packages

Although bundles and packages are sometimes referred to interchangeably, they actually represent very distinct concepts:

- A **package** is any directory that the Finder presents to the user as if it were a single file.
- A **bundle** is a directory with a standardized hierarchical structure that holds executable code and the resources used by that code.

Packages provide one of the fundamental abstractions that makes Mac OS X easy to use. If you look at an application or plug-in on your computer, what you are actually looking at is a directory. Inside the package directory are the code and resource files needed to make the application or plug-in run. When you interact with the package directory, however, the Finder treats it like a single file. This behavior prevents casual users from making changes that might adversely affect the contents of the package. For example, it prevents users from rearranging or deleting resources or code modules that might prevent an application from running correctly.

Note: Even though packages are treated as opaque files by default, it is still possible for users to view and modify their contents. On the contextual menu for package directories is a Show Package Contents command. Selecting this command displays a new Finder window set to the top level of the package directory. The user can use this window to navigate the package's directory structure and make changes as if it were a regular directory hierarchy.

Whereas packages are there to improve the user experience, bundles are geared more toward helping developers package their code and to helping the operating system access that code. Bundles define the basic structure for organizing the code and resources associated with your software. The presence of this structure also helps facilitate important features such as localization. The exact structure of a bundle depends on whether you are creating an application, framework, or plug-in. It also depends on other factors such as the target platform and the type of plug-in.

The reason bundles and packages are sometimes considered to be interchangeable is that many types of bundles are also packages. For example, applications and loadable bundles are packages because they are usually treated as opaque directories by the system. However, not all bundles are packages and vice versa.

How the System Identifies Bundles and Packages

The Finder considers a directory to be a package if any of the following conditions are true:

- The directory has a known filename extension: `.app`, `.bundle`, `.framework`, `.plugin`, `.kext`, and so on.
- The directory has an extension that some other application claims represents a package type; see "Document Packages."
- The directory has its package bit set.

The preferred way to specify a package is to give the package directory a known filename extension. For the most part, Xcode takes care of this for you by providing templates that apply the correct extension. All you have to do is create an Xcode project of the appropriate type.

Most bundles are also packages. For example, applications and plug-ins are typically presented as a single file by the Finder. However, this is not true for all bundle types. In particular, a framework is a type of bundle that is treated as a single unit for the purposes of linking and runtime usage, but framework directories are transparent so that developers can view the header files and other resources they contain.

About Bundle Display Names

Display names give the user some control over how bundles and packages appear in the Finder without breaking clients that rely on them. Whereas a user can rename a file freely, renaming an application or framework might cause related code modules that refer to the application or framework by name to break. Therefore, when the user changes the name of a bundle, the change is superficial only. Rather than change the bundle name in the file system, the Finder associates a separate string (known as the **display name**) with the bundle and displays that string instead.

Display names are for presentation to the user only. You never use display names to open or access directories in your code, but you do use them when displaying the name of the directory to the user. By default, a bundle's display name is the same as the bundle name itself. However, the system may alter the default display name in the following cases:

- If the bundle is an application, the Finder hides the `.app` extension in most cases.
- If the bundle supports localized display names (and the user has not explicitly changed the bundle name), the Finder displays the name that matches the user's current language settings.

Although the Finder hides the `.app` extension for applications most of the time, it may display it to prevent confusion. For example, if the user changes the name of an application and the new name contains another filename extension, the Finder shows the `.app` extension to make it clear that the bundle is an application. For example, if you were to add the `.mov` extension to the `Chess` application, the Finder would display `Chess.mov.app` to prevent users from thinking `Chess.mov` is a QuickTime file.

For more information about display names and specifying localized bundle names, see *File System Overview*.

The Advantages of Bundles

Bundles provide the following advantages for developers:

- Because bundles are directory hierarchies in the file system, a bundle just contains files. Therefore, you can use all of the same file-based interfaces to open your bundle resources as you do to open other types of files.
- The bundle directory structure makes it easy to support multiple localizations. 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.
- Users can install, relocate, and remove bundles simply by dragging them around in the Finder.
- Bundles that are also packages, and are therefore treated as opaque files, are less susceptible to accidental user modifications, such as removal, modification, or renaming of critical resources.
- A bundle can support multiple chip architectures (PowerPC, Intel) and different address space requirements (32-bit/64-bit). It can also support the inclusion of specialized executables (for example, libraries optimized for a particular set of vector instructions).
- Most (but not all) executable code can be bundled. Applications, frameworks (shared libraries), and plug-ins all support the bundle model. Static libraries, dynamic libraries, shell scripts, and UNIX command line tools do not use the bundle structure.
- A bundled application can run directly from a server. No special shared libraries, extensions, and resources need to be installed on the local system.

Types of Bundles

Although all bundles support the same basic features, there are variations in the way you define and create bundles that define their intended usage:

- **Application** - An application bundle manages the code and resources associated with a launchable process. The exact structure of this bundle depends on the platform (iPhone OS or Mac OS X) that you are targeting. For information about the structure of application bundles, see "Application Bundles."
- **Frameworks** - A framework bundle manages a dynamic shared library and its associated resources, such as header files. An application can link against one or more frameworks to take advantage of the code they contain. For information about the structure of framework bundles, see "Anatomy of a Framework Bundle."
- **Plug-Ins** - Mac OS X supports plug-ins for many system features. Plug-ins are a way for an application to load custom code modules dynamically. The following list identifies some of the key types of plug-ins you might want to develop:
    - **Custom plug-ins** are plug-ins you define for your own purposes; see "Anatomy of a Loadable Bundle."
    - **Image Unit plug-ins** add custom image-processing behaviors to the Core Image technology; see *Image Unit Tutorial*.
    - **Interface Builder plug-ins** contain custom objects that you want to integrate into Interface Builder's library window; see *Interface Builder Plug-In Programming Guide*.
    - **Preference Pane plug-ins** define custom preferences that you want to integrate into the System Preferences application; see *Preference Pane Programming Guide*.
    - **Quartz Composer plug-ins** define custom patches for the Quartz Composer application; see *Quartz Composer Custom Patch Programming Guide*.
    - **Quick Look plug-ins** support the display of custom document types using Quick Look; see *Quick Look Programming Guide*.
    - **Spotlight plug-ins** support the indexing of custom document types so that those documents can be searched by the user; see *Spotlight Importer Programming Guide*.
    - **Sync Schema plug-ins** identify custom information that can be synchronized with the system; see *Sync Services Programming Guide*.
    - **WebKit plug-ins** extend the content types supported by common web browsers; see *WebKit Plug-In Programming Topics*.
    - **Widgets** add new HTML-based applications to Dashboard; see *Dashboard Programming Topics*.

Although document formats can leverage the bundle structure to organize their contents, documents are generally not considered bundles in the purest sense. A document that is implemented as a directory and treated as an opaque type is considered to be a document package, regardless of its internal format. For more information about document packages, see "Document Packages."

Creating a Bundle

For the most part, you do not create bundles or packages manually. When you create a new Xcode project (or add a target to an existing project), Xcode automatically creates the required bundle structure when needed. For example, the application, framework, and loadable bundle targets all have associated bundle structures. When you build any of these targets, Xcode automatically creates the corresponding bundle for you.

Note: Some Xcode targets (such as shell tools and static libraries) do not result in the creation of a bundle or package. This is normal and there is no need to create bundles specifically for these target types. The resulting binaries generated for those targets are intended to be used as is.

If you use make files (instead of Xcode) to build your projects, there is no magic to creating a bundle. A bundle is just a directory in the file system with a well-defined structure and a specific filename extension added to the end of the bundle directory name. As long as you create the top-level bundle directory and structure the contents of your bundle appropriately, you can access those contents using the programmatic support for accessing bundles. For more information on how to structure your bundle directory, see "Bundle Structures."

Programmatic Support for Accessing Bundles

Programs that refer to bundles, or are themselves bundled, can take advantage of interfaces in Cocoa and Core Foundation to access the contents of a bundle. Using these interfaces you can find bundle resources, get information about the bundle's configuration, and load executable code. In Objective-C applications, you use the `NSBundle` class to get and manage bundle information. For C-based applications, you can use the functions associated with the `CFBundleRef` opaque type to manage a bundle.

Note: Unlike many other Core Foundation and Cocoa types, `NSBundle` and `CFBundleRef` are not toll-free bridged data types and cannot be used interchangeably. However, you can extract the bundle path information from either object and use it to create the other.

For information about how to use the programmatic support in Cocoa and Core Foundation to access bundles, see "Accessing a Bundle's Contents."

Guidelines for Using Bundles

Bundles are the preferred organization mechanism for software in Mac OS X and iPhone OS. The bundle structure lets you group executable code and the resources to support that code in one place and in an organized way. The following guidelines offer some additional advice on how to use bundles:

- Always include an information-property list (`Info.plist`) file in your bundle. Make sure you include the keys recommended for your bundle type. For a list of all keys you can include in this file, see *Runtime Configuration Guidelines*.
- If an application cannot run without a specific resource file, include that file inside the application bundle. Applications should always include all of the images, strings files, localizable resources, and plug-ins that they need to operate. Noncritical resources should similarly be stored inside the application bundle whenever possible but may be placed outside the bundle if needed. For more information about the bundle structure of applications, see "Application Bundles."
- If you plan to load C++ code from a bundle, you might want to mark the symbols you plan to load as `extern "C"`. Neither `NSBundle` nor the Core Foundation `CFBundleRef` functions know about C++ name mangling conventions, so marking your symbols this way can make it much easier to identify them later.
- You cannot use the `NSBundle` class to load Code Fragment Manager (CFM) code. If you need to load CFM-based code, you must use the functions for the `CFBundleRef` or `CFPlugInRef` opaque types. You may load CFM-based plugins from a Mach-O executable using this technique.
- You should always use the `NSBundle` class (as opposed to the functions associated with the `CFBundleRef` opaque type) to load any bundle containing Java code.
- When loading bundles containing Objective-C code, you may use either the `NSBundle` class or the functions associated with the `CFBundleRef` opaque type in Mac OS X v10.5 and later, but there are differences in behavior for each. If you use the Core Foundation functions to load a plug-in or other loadable bundle (as opposed to a framework or dynamic shared library), the functions load the bundle privately and bind its symbols immediately; if you use `NSBundle`, the bundle is loaded globally and its symbols are bound lazily. In addition, bundles loaded using the `NSBundle` class cause the generation of `NSBundleDidLoadNotification` notifications, whereas those loaded using the Core Foundation functions do not.

NextPrevious

https://web.archive.org/web/20100525020133/http://developer.apple.com/mac/library/documentation/coref
oundation/conceptual/CFBundles/Introduction/Introduction.html

Download IconCompanion File PDF

Next

Introduction

Bundles are a fundamental technology in Mac OS X and iPhone OS that are used to encapsulate code and resources. Bundles simplify the developer experience by providing known locations for needed resources while alleviating the need to create compound binary files. Instead, bundles use directories and files to provide a more natural type of organization—one that can also be modified easily both during development and after deployment.

To support bundles, both Cocoa and Core Foundation provide programming interfaces for accessing the contents of bundles. Because bundles use an organized structure, it is important that all developers understand the fundamental organizing principles of bundles. This document provides you with the foundation for understanding how bundles work and for how you use them during development to access your resource files.

Organization of This Document

This document contains the following chapters:

- "About Bundles" introduces the concept of bundles and packages and how they are used by the system.
- "Bundle Structures" describes the structure and contents of the standard bundle types.
- "Accessing a Bundle's Contents" shows you how to use the Cocoa and Core Foundation interfaces to get information about a bundle and its contents.
- "Document Packages" describes the notion of document packages (which are loosely related to bundles) and how you use them.

See Also

Although the information in this document applies to all types of bundles, if you are working with more specialized types of bundles (such as frameworks and plug-ins), you should also consult the following documents:

- *Framework Programming Guide* provides detailed information about creating and using custom frameworks.
- *Code Loading Programming Topics for Cocoa* provides information about writing plug-ins using the Objective-C language.
- *Plug-ins* provides information about writing plug-ins using the C language.

Next

https://web.archive.org/web/20100504092324/http://developer.apple.com/mac/library/navigation/

Mac OS X Reference Library

Search Mac Reference Library

Overview | Getting Started | Required Reading | Featured

Learn the basics about Mac OS development by reading these documents.

- Mac OS X Technology Overview
- Cocoa Application Tutorial
- Learning Objective-C: A Primer
- A Tour of Xcode

You will see additional lists of Getting Started documents in the upper-right corners of topic pages. Each of these lists is specific to the topic at hand.

Documents

Loading...

Did this document help you? Yes | It's good, but... | Not helpful...

https://web.archive.org/web/20100414095727/http://developer.apple.com:80/mac/

Log in to get the most out of the Mac Dev Center.

Log in

The Mac Dev Center provides access to technical resources and information to assist you in developing with the latest technologies in Mac OS X. Log in with the Apple ID and password you used to register as an Apple Developer, or register for free today.

Developing for Mac OS X Snow Leopard

Search Mac Reference Library

Technical Documentation

Sample Code
Each Sample Code project is a buildable and executable source example of how to accomplish a task for a specific Apple technology.

Mac OS X Reference Library
The Reference Library is a comprehensive collection of Apple technical resources, including Articles, Guides, Reference, Release, Notes, Sample Code, Technical Notes, and Technical Q&As.

Featured Technical Content

- Mac OS X Technology Overview
- What's New in Mac OS X
- Concurrency Programming Guide
- What's New in Xcode
- Introducing Blocks and Grand Central Dispatch

To download Xcode and access a range of development videos, log in with the Apple ID and password you used to register as an Apple Developer, or register for free today.

Download Xcode 3.2.2 and iPhone SDK 3.2 for Snow Leopard
Includes the Xcode IDE, Dashcode, Instruments, Interface Builder, and the rest of the developer tools.

Development Videos
Watch Apple engineers and experts to see how you can start using technology from Mac OS X in your development process today.

Bug Reporting
Use the Apple Bug Reporter to submit and track bug reports and enhancement requests.

Mac Developer Program

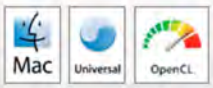
Download Mac OS X Snow Leopard
Mac Developer Program Members, download the latest build of the next generation of the world's most advanced operating system.
Download now

Join the Mac Developer Program
The Mac Developer Program offers a range of technical resources and support for developers and IT professionals innovating with Mac OS X.
Learn more

Develop for Snow Leopard
The core technologies in Snow Leopard unleash the power of today's advanced hardware technology and prepare Mac OS X for future innovation.

Developer Tools & Technologies
Read about why you'll love to develop on Apple Platforms.

Software Licensing & Trademarks
Use Apple Software, technologies and trademarks in your product.



WWDC Session Videos
Session videos from the Apple Worldwide Developers Conference 2009 are available for purchase.

Book Details

By Scott Meyers , Mike Lee
ISBN13: 978-1-4302-1946-0
ISBN10: 1-4302-1946-7
632 pp.
Published Sep 2009
Print Book Price: $34.99
eBook Price: $24.49

Buy eBook
Buy Print Book
SHARE

Book Resources

Submit/View Errata

My eBook

My Account
Download eBook(s)

Learn Mac OS X Snow Leopard

You're smart and savvy, but also busy. This comprehensive guide to Apple's latest version of Mac OS X 10.6, Snow Leopard, gives you everything you need to know to live a happy, productive Mac life. *Learn Mac OS X Snow Leopard* will have you up and connected lickity-split. With a minimum of overhead and a maximum of useful information, you'll cover a lot of ground in the time it takes other books to get you plugged in.

If this isn't you're first experience with Mac OS X, skip right to the "What's New in Snow Leopard" sections. You may also find yourself using this book as a quick refresher course or a way to learn new Mac skills you've never tried before.

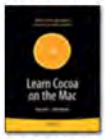
What you'll learn

- Discover all the ins and outs of the Finder and Snow Leopard's streamlined workflow.
- Use the latest features of Apple's built-in applications, including Mail, Safari, iCal, Address Book, iChat, Preview, and more.
- Learn about Snow Leopard's improved security and reliability, and how to take full advantage of the connected world and a wealth of new mobile devices.
- Administer your computer and network for yourself, your family, or your business.
- Work with add-on devices via direct connection or wirelessly.
- Master effective strategies for data backup, recovery, and security.
- Explore all of Apple's improved iLife applications, including iTunes, iPhoto, and iMovie.
- Delve into more advanced topics, such as how to take advantage of the Darwin subsystem in Leopard, how to run multiple operating systems on your Mac, and how to cooperate with other operating systems (and help them cooperate with you).
- Get started with your own Mac OS X development in Snow Leopard

Who is this book for?

New Mac users, existing Mac users upgrading from older versions of Mac OS X, and PC users making the switch to Macs. These people share a common desire to learn stuff fast and keep learning! Because this book goes into greater depth than your average Mac OS X guide, it is also excellent for small business owners, user support personnel, and system administrators. There's even an introduction to Mac OS X development for nascent programmers and the DIY crowd.

Related Titles



Learn Cocoa on the Mac

Learn C on the Mac

Learn Objective-C on the Mac

Mac for Linux Geeks

Beginning iPhone Development: Exploring the iPhone SDK

Foundations of Mac OS X Leopard Security

Taking Your iPod touch to the Max

Taking Your iPhone to the Max

Author Information

Scott Meyers

Scott Meyers has worked in and around the computer industry, beginning as an Apple Sales Specialist and Consultant over 12 years ago, he has since moved on to various other jobs including Web Design and Development, Editing books on Web Development, Open Source and Apple Technology, and Marketing. He is a Select ADC (Apple Developers Connection) Member and a huge fan of Mac OS X which brings together his love of the Apple's traditionally best of class GUI and applications with the unrivaled power of Unix and Open Source technologies and applications.

Scott lives outside of Indianapolis, Indiana with his wife, two kids, and a cat and a dog. When not working or writing he enjoys photography, playing guitar, and building amplifiers.

Mike Lee

Mike Lee, the world's toughest programmer, is the founder and CEO of United Lemur, a philanthropic revolution disguised as a software company. Mike has had a role in creating many popular iPhone applications, including Obama '08, Tap Tap Revenge, Twinkle, and Jott.

Prior to iPhone, Mike cut his teeth — and won an Apple Design Award — at Seattle-based Delicious Monster Software. Mike is originally from Honolulu, is a popular blogger and occasional pundit, and has been seen on Twitter as @bmf.

Mike and his wife are originally from Honolulu, but currently live in Silicon Valley where they are raising two cats. Mike's hobbies include weightlifting, single malts, and fire.

Mike can be contacted at mike@unitedlemur.org.

EXHIBIT B

https://web.archive.org/web/20101113134432mp_/http://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/OSX_Technology_Overview/OSX_Technology_Overview.pdf

Mac OS X Technology Overview

General

2009-08-14

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

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction to Mac OS X Technology Overview

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

Who Should Read This Document

Mac OS X Technology Overview is an essential guide for anyone looking to develop software for Mac OS X. It provides an overview of the technologies and tools that have an impact on the development process and provides links to relevant documents and other sources of information. You should use this document to do the following:

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

This document does not provide information about user-level system features or about features that have no impact on the software development process.

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

Organization of This Document

This document has the following chapters and appendixes:

- "Mac OS X System Overview" (page 17) provides background information for understanding the terminology and basic development environment of Mac OS X. It also provides a high-level overview of the Mac OS X system architecture.
- "Darwin and Core Technologies" (page 21) describes the technologies that comprise the Darwin environment along with other key technologies that are used throughout the system.
- "Graphics and Multimedia Technologies" (page 43) describes the graphics foundations of the system, including the technologies you use for drawing to the screen and for creating audio and video content.

- "Application Technologies" (page 57) describes the development environments (like Carbon and Cocoa) and individual technologies (like Address Book) that you use to create your applications.
- "User Experience" (page 71) describes the technologies that your application should use to provide the best user experience for the platform. This chapter also describes some of the system technologies with which your software interacts to create that experience.
- "Software Development Overview" (page 79) describes the types of software you can create for Mac OS X and when you might use each type.
- "Choosing Technologies to Match Your Design Goals" (page 93) provides tips and guidance to help you choose the technologies that best support the design goals of your application.
- "Porting Tips" (page 103) provides starter advice for developers who are porting applications from Mac OS 9, Windows, and UNIX platforms.
- "Command Line Primer" (page 109) provides an introduction to the command-line interface for developers who have never used it before.
- "Mac OS X Frameworks" (page 115) describes the frameworks you can use to develop your software. Use this list to find specific technologies or to find when a given framework was introduced to Mac OS X.
- "Mac OS X Developer Tools" (page 127) provides an overview of the available applications and command-line tools you can use to create software for Mac OS X.

Getting the Xcode Tools

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

For additional information about the tools available for working with Mac OS X and its technologies, see "Mac OS X Developer Tools" (page 127).

Reporting Bugs

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

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

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

See Also

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

For information about new features introduced in different versions of Mac OS X, see *What's New In Mac OS X*.

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

Developer Documentation

When you install Xcode, the installer places the tools you need for development as well as sample code and developer documentation on your local hard drive. The default installation directory for Xcode is `/Developer` but in Mac OS X v10.5 and later you can specify a custom installation directory if desired. (This document uses the term *<Xcode>* to represent the root directory of your Xcode installation.) The Installer application puts developer documentation into the following locations:

- **General documentation**. Most documentation and sample code is installed in the *<Xcode>*`/Documentation/DocSets` directory. All documents are available in HTML format, which you can view from any web browser. To view the documentation, open the Xcode IDE and choose Help > Show Documentation Window.
- **Additional sample code**. Some additional sample programs are installed in *<Xcode>*`/Examples`. These samples demonstrate different tasks involving Mac OS X technologies.

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

Information on BSD

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

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

- Apple's Open Source page (http://developer.apple.com/opensource/)
- The FreeBSD project (http://www.freebsd.org)
- The NetBSD project (http://www.netbsd.org)
- The OpenBSD project (http://www.openbsd.org)

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

Darwin and Open Source Development

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

Darwin consists of the Mac OS X kernel environment, BSD libraries, and BSD command environment. For more information about Darwin and what it contains, see "Kernel and Drivers" (page 21). For detailed information about the kernel environment, see *Kernel Programming Guide*.

Information about the Darwin open source efforts is available at http://developer.apple.com/darwin/ and at http://www.macosforge.org/.

Other Information on the Web

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

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

Mac OS X System Overview

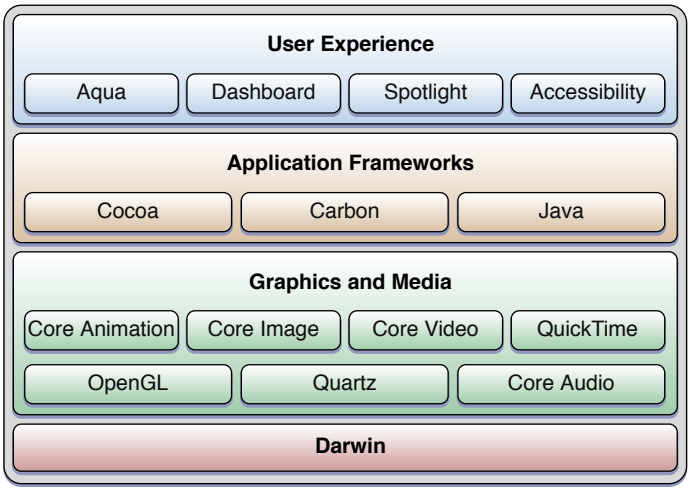
This chapter provides a high-level introduction to Mac OS X, describing its overall architecture and development tools support. The goal of this chapter is to orient you to the Mac OS X operating system and to give you a reference point from which to explore the available tools and technologies described throughout this document. Developers who are already familiar with the Mac OS X system architecture and technologies may want to skip this chapter.

Note: For a listing of commonly used Mac OS X terms, see "Glossary" (page 159).

A Layered Approach

The implementation of Mac OS X can be viewed as a set of layers. At the lower layers of the system are the fundamental services on which all software relies. Subsequent layers contain more sophisticated services and technologies that build on (or complement) the layers below. Figure 1-1 provides a graphical view of this layered approach, highlighting a few of the key technologies found in each layer of Mac OS X.

Figure 1-1 Layers of Mac OS X



The bottom layer consists of the core environment layer, of which Darwin is the most significant component. Darwin is the name given to the FreeBSD environment that comprises the heart of Mac OS X. FreeBSD is a variant of the Berkeley Software Distribution UNIX environment, which provides a secure and stable foundation for building software. Included in this layer are the kernel environment, device drivers, security support, interprocess communication support, and low-level commands and services used by all programs on the system. Besides Darwin, this layer contains several core services and technologies, many of which are simply higher-level wrappers for the data types and functions in the Darwin layer. Among the available core services

are those for doing collection management, data formatting, memory management, string manipulation, process management, XML parsing, stream-based I/O, and low-level network communication. For details about the technologies in this layer, see "Darwin and Core Technologies" (page 21).

The Graphics and Media layer implements specialized services for playing audio and video and for rendering 2D and 3D graphics. One of the key technologies in this layer is Quartz, which provides the main rendering environment and window management support for Mac OS X applications. QuickTime is Apple's technology for displaying video, audio, virtual reality, and other multimedia-related information. Apple's core technologies, including Core Image, Core Video, Core Animation, and Core Audio, provide advanced behavior for different types of media. OpenGL is an implementation of the industry-standard application programming interface (API) for rendering graphics and is used both as a standalone technology and as an underlying technology for accelerating all graphics operations. For details about the technologies in this layer, see "Graphics and Multimedia Technologies" (page 43).

The Application Frameworks layer embodies the technologies for building applications. At the heart of this layer are the basic environments used to develop applications: Cocoa, Carbon, Java, and others. Each environment is designed to provide a level of familiarity to certain types of developers. For example, Cocoa and Java provide object-oriented environments using the Objective-C and Java languages while Carbon provides a C-based environment. This layer also contains numerous supporting technologies, such as Core Data, Address Book, Image Services, Keychain Services, Launch Services, HTML rendering, and many others. These technologies provide advanced user features and can be used to shorten your overall development cycle. For details about the technologies in this layer, see "Application Technologies" (page 57).

The User Experience layer identifies the methodologies, technologies, and applications that make Mac OS X software unique. Apple provides countless technologies to implement the overall user experience. Many of these technologies simply work, but some require interactions with the software you create. Understanding what interactions are expected of your software can help you integrate it more smoothly into the Mac OS X ecosystem. For details about the technologies in this layer, see "User Experience" (page 71).

The Advantage of Layers

The nice thing about the Mac OS X layered design is that writing software in one layer does not preclude you from using technologies in other layers. Mac OS X technologies were built to interoperate with each other whenever possible. In cases where a given technology is unsuitable, you can always use a different technology that is suitable. For example, Cocoa applications can freely use Carbon frameworks and BSD function calls. Similarly, Carbon applications can use Objective-C based frameworks in addition to other object-oriented and C-based frameworks. Of course, in the case of Carbon, you might have to set up some Cocoa-specific structures before creating any Cocoa objects, but doing so is relatively trivial.

Although you may feel more comfortable sticking with your chosen development environment, there are advantages to straying outside of that environment. You might find that technologies in other layers offer better performance or more flexibility. For example, using the POSIX interfaces in the Darwin layer might make it easier to port your application to other platforms that conform to the POSIX specification. Having access to technologies in other layers gives you options in your development process. You can pick and choose the technologies that best suit your development needs.

Developer Tools

Mac OS X provides you with a full suite of free developer tools to prototype, compile, debug, and optimize your applications. At the heart of Apple's developer tools solution is **Xcode**, Apple's integrated development environment. You use Xcode to organize and edit your source files, compile and debug your code, view documentation, and build all manner of software products.

In addition to the Xcode application, Mac OS X also provides you with a wide selection of open source tools, such as the GNU Compiler Collection (GCC), which you use to build Mach-O programs, the native binary format of Mac OS X. If you are used to building programs from the command line, all of the familiar tools are there for you to use, including makefiles, the gdb debugger, analysis tools, performance tools, source-code management tools, and many other code utilities.

Mac OS X also provides many other tools to make the development process easier:

- **Interface Builder** lets you design your application's user interface graphically and save those designs as resource files that you can load into your program at runtime.
- **Instruments** is a powerful performance analysis and debugging tool that lets you peer into your code as it's running and gather important metrics about what it is doing.
- **Shark** is an advanced statistical analysis tool that turns your code inside out to help you find any performance bottlenecks.
- **PackageMaker** helps you build distributable packages for delivering your software to customers.
- Mac OS X includes several OpenGL tools to help you analyze the execution patterns and performance of your OpenGL rendering calls.
- Mac OS X supports various scripting languages, including Perl, Python, Ruby, and others.
- Mac OS X includes tools for creating and working with Java programs.

Installing the developer tools also installs the header files and development directories you need to develop software. For information on how to get the developer tools, see "Getting the Xcode Tools" (page 14). For more information about the tools themselves, see "Mac OS X Developer Tools" (page 127).

Darwin and Core Technologies

This chapter summarizes the fundamental system technologies and facilities that are available to developers in Mac OS X. The Darwin layer of Mac OS X comprises the kernel, drivers, and BSD portions of the system and is based primarily on open source technologies. Mac OS X extends this low-level environment with several core infrastructure technologies that make it easier for you to develop software.

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

Kernel and Drivers

Beneath the appealing, easy-to-use interface of Mac OS X is a rock-solid, UNIX-based foundation that is engineered for stability, reliability, and performance. The kernel environment is built on top of Mach 3.0 and provides high-performance networking facilities and support for multiple, integrated file systems.

The following sections describe some of the key features of the kernel and driver portions of Darwin. For pointers to more information about the kernel environment, see *Getting Started with Darwin*.

Mach

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

- **Protected memory**. The stability of an operating system should not depend on all executing applications being good citizens. Even a well-behaved process can accidentally write data into the address space of the system or another process, which can result in the loss or corruption of data or even precipitate system crashes. Mach ensures that an application cannot write in another application's memory or in the operating system's memory. By walling off applications from each other and from system processes, Mach makes it virtually impossible for a single poorly behaved application to damage the rest of the system. Best of all, if an application crashes as the result of its own misbehavior, the crash affects only that application and not the rest of the system.
- **Preemptive multitasking**. With Mach, processes share the CPU efficiently. Mach watches over the computer's processor, prioritizing tasks, making sure activity levels are at the maximum, and ensuring that every task gets the resources it needs. It uses certain criteria to decide how important a task is and therefore how much time to allocate to it before giving another task its turn. Your process is not dependent on another process yielding its processing time.

- **Advanced virtual memory**. In Mac OS X, virtual memory is "on" all the time. The Mach virtual memory system gives each process its own private virtual address space. For 32-bit applications, this virtual address space is 4 GB. For 64-bit applications, the theoretical maximum is approximately 18 exabytes, or 18 billion billion bytes. Mach maintains address maps that control the translation of a task's virtual addresses into physical memory. Typically only a portion of the data or code contained in a task's virtual address space resides in physical memory at any given time. As pages are needed, they are loaded into physical memory from storage. Mach augments these semantics with the abstraction of memory objects. Named memory objects enable one task (at a sufficiently low level) to map a range of memory, unmap it, and send it to another task. This capability is essential for implementing separate execution environments on the same system.
- **Real-time support**. This feature guarantees low-latency access to processor resources for time-sensitive media applications.

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

64-Bit Kernel

Mac OS X v10.6 and later contains a 64-bit kernel. Although Mac OS X allows a 32-bit kernel to run 64-bit applications, a 64-bit kernel provides several benefits:

- The kernel can support large memory configurations more efficiently.
- The maximum size of the buffer cache is increased, potentially improving I/O performance.
- Performance is improved when working with specialized networking hardware that emulates memory mapping across a wire or with multiple video cards containing over 2 GB of video RAM.

Because a 64-bit kernel does not support 32-bit drivers and kexts, those items must be built for 64-bit. Fortunately, for most drivers, this is usually not as difficult as you might think. For the most part, transitioning a driver to be 64-bit capable is just like transitioning any other piece of code. For details about how to make the transition, including what things to check for in your code, see *64-Bit Transition Guide*.

Device-Driver Support

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

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

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

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

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

File-System Support

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

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

Because of its multiple application environments and the various kinds of devices it supports, Mac OS X handles file data in many standard volume formats. Table 2-1 lists the supported formats.

Table 2-1 Supported local volume formats

Volume format	Description
Mac OS Extended Format	Also called HFS (hierarchical file system) Plus, or HFS+. This is the default root and booting volume format in Mac OS X. This extended version of HFS optimizes the storage capacity of large hard disks by decreasing the minimum size of a single file.
Mac OS Standard Format	Also called hierarchical file system, or HFS. This is the volume format in Mac OS systems prior to Mac OS 8.1. HFS (like HFS+) stores resources and data in separate forks of a file and makes use of various file attributes, including type and creator codes.
UDF	Universal Disk Format, used for hard drives and optical disks, including most types of CDs and DVDs. Mac OS X v10.4 supports UDF revisions 1.02 through 1.50 (although you cannot write out Finder Info, resource forks, and other extended attributes in these revisions). Mac OS X v10.5 and later supports reading UDF revisions 1.02 through 2.60 on both block devices and most optical media, and it supports writing to block devices and to DVD-RW and DVD+RW media using UDF 2.00 through 2.50 (except for mirrored metadata partions in 2.50). You can find the UDF specification at http://www.osta.org.
ISO 9660	The standard format for CD-ROM volumes.

Volume format	Description
NTFS	The NT File System, used by Windows computers. Mac OS X can read NTFS-formatted volumes but cannot write to them.
UFS	UNIX File System is a flat (that is, single-fork) disk volume format, based on the BSD FFS (Fast File System), that is similar to the standard volume format of most UNIX operating systems; it supports POSIX file-system semantics, which are important for many server applications. Although UFS is supported in Mac OS X, its use is discouraged.
MS-DOS (FAT)	Mac OS X supports the FAT file systems used by many Windows computers. It can read and write FAT-formatted volumes.

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

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

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

Table 2-2 Supported network file-sharing protocols

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

Network Support

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

The Mac OS X network protocol stack is based on BSD. The extensible architecture provided by network kernel extensions, summarized in "Networking Extensions" (page 28), facilitates the creation of modules implementing new or existing protocols that can be added to this stack.

Standard Network Protocols

Mac OS X provides built-in support for a large number of network protocols that are standard in the computing industry. Table 2-3 summarizes these protocols.

Table 2-3 Network protocols

Protocol	Description
802.1x	802.1x is a protocol for implementing port-based network access over wired or wireless LANs. It supports a wide range of authentication methods, including TLS, TTLS, LEAP, MDS, and PEAP (MSCHAPv2, MD5, GTC).
DHCP and BOOTP	The Dynamic Host Configuration Protocol and the Bootstrap Protocol automate the assignment of IP addresses in a particular network.
DNS	Domain Name Services is the standard Internet service for mapping host names to IP addresses.
FTP and SFTP	The File Transfer Protocol and Secure File Transfer Protocol are two standard means of moving files between computers on TCP/IP networks. (SFTP support was added in Mac OS X version 10.3.)
HTTP and HTTPS	The Hypertext Transport Protocol is the standard protocol for transferring webpages between a web server and browser. Mac OS X provides support for both the insecure and secure versions of the protocol.
LDAP	The Lightweight Directory Access Protocol lets users locate groups, individuals, and resources such as files and devices in a network, whether on the Internet or on a corporate intranet.
NBP	The Name Binding Protocol is used to bind processes across a network.
NTP	The Network Time Protocol is used for synchronizing client clocks.
PAP	The Printer Access Protocol is used for spooling print jobs and printing to network printers.
PPP	For dialup (modem) access, Mac OS X includes PPP (Point-to-Point Protocol). PPP support includes TCP/IP as well as the PAP and CHAP authentication protocols.
PPPoE	The Point-to-Point Protocol over Ethernet protocol provides an Ethernet-based dialup connection for broadband users.
S/MIME	The Secure MIME protocol supports encryption of email and the attachment of digital signatures to validate email addresses. (S/MIME support was added in Mac OS X version 10.3.)
SLP	Service Location Protocol is designed for the automatic discovery of resources (servers, fax machines, and so on) on an IP network.
SOAP	The Simple Object Access Protocol is a lightweight protocol for exchanging encapsulated messages over the web or other networks.

Protocol	Description
SSH	The Secure Shell protocol is a safe way to perform a remote login to another computer. Session information is encrypted to prevent unauthorized snooping of data.
TCP/IP and UDP/IP	Mac OS X provides two transmission-layer protocols, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), to work with the network-layer Internet Protocol (IP). (Mac OS X 10.2 and later includes support for IPv6 and IPSec.)
XML-RPC	XML-RPC is a protocol for sending remote procedure calls using XML over the web.

Apple also implements a number of file-sharing protocols; see Table 2-2 (page 24) for a summary of these protocols.

Legacy Network Services and Protocols

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

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

These protocols are provided to support legacy applications, such as those running in the Classic environment. You should never use these protocols for any active development. Instead, you should use newer networking technologies such as CFNetwork.

Network Technologies

Mac OS X supports the network technologies listed in Table 2-4.

Table 2-4 Network technology support

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

Technology	Description
Wireless	Supports the 802.11b, 802.11g, and 802.11n wireless network technology using AirPort and AirPort Extreme.

Routing and Multihoming

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

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

Zero-Configuration Networking

Introduced in Mac OS X version 10.2, Bonjour is Apple's implementation of zero-configuration networking. Bonjour enables the dynamic discovery of computer services over TCP/IP networks without the need for any complex user configuration of the associated hardware. Bonjour helps to connect computers and other electronic devices by providing a mechanism for them to advertise and browse for network-based services. See "Bonjour" (page 61) for more information.

NetBoot

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

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

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

Personal Web Sharing

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

Networking Extensions

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

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

For information on how to write an NKE, see *Network Kernel Extensions Programming Guide*.

Network Diagnostics

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

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

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

BSD

Integrated with Darwin is a customized version of the Berkeley Software Distribution (BSD) operating system (currently FreeBSD 5). Darwin's implementation of BSD includes much of the POSIX API, which higher-level applications can also use to implement basic application features. BSD serves as the basis for the file systems and networking facilities of Mac OS X. In addition, it provides several programming interfaces and services, including:

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

Note: For more information about the FreeBSD operating system, go to http://www.freebsd.org/. For more information about the boot process of Mac OS X, including how it launches the daemons used to implement key BSD services, see *System Startup Programming Topics*.

The following sections describe some of the key features of the BSD Layer of Mac OS X.

Caching API

Introduced in Mac OS X v10.6, the `libcache` API is a low-level purgeable caching API. Aggressive caching is an important technique in maximizing application performance. However, when caching demands exceed available memory, the system must free up memory as necessary to handle new demands. Typically, this means paging cached data to and from relatively slow storage devices, sometimes even resulting in system-wide performance degradation. Your application should avoid potential paging overhead by actively managing its data caches, releasing them as soon as it no longer needs the cached data.

In the wider system context, your application can now also help by creating caches that the operating system can simply purge on a priority basis as memory pressure necessitates. Mac OS X v10.6 includes the `libcache` library and Foundation framework's `NSCache` class to create these purgeable caches.

For more information about the functions of the `libcache` library, see *libcache Reference*. For more information about the `NSCache` class, see *NSCache Class Reference*.

Scripting Support

Darwin includes all of the scripting languages commonly found in UNIX-based operating systems. In addition to the scripting languages associated with command-line shells (such as `bash` and `csh`), Darwin also includes support for Perl, Python, Ruby, and others.

In Mac OS X v10.5, Darwin added support for several new scripting features. In addition to adding support for Ruby on Rails, Mac OS X also added scripting bridges to the Objective-C classes of Cocoa. These bridges let you use Cocoa classes from within your Python and Ruby scripts. For information about using these bridges, see *Ruby and Python Programming Topics for Mac OS X*.

For information about scripting tools, see "Scripting Tools" (page 153). For information on using command-line shells, see "Command Line Primer" (page 109).

Threading Support

Mac OS X provides full support for creating multiple preemptive threads of execution inside a single process. Threads let your program perform multiple tasks in parallel. For example, you might create a thread to perform some lengthy calculations in the background while a separate thread responds to user events and updates the windows in your application. Using multiple threads can often lead to significant performance improvements in your application, especially on computers with multiple CPU cores. Multithreaded programming is not without its dangers though and requires careful coordination to ensure your application's state does not get corrupted.

All user-level threads in Mac OS X are based on POSIX threads (also known as pthreads). A pthread is a lightweight wrapper around a Mach thread, which is the kernel implementation of a thread. You can use the pthreads API directly or use any of the threading packages offered by Cocoa, Carbon, or Java, all of which are implemented using pthreads. Each threading package offers a different combination of flexibility versus ease-of-use. All offer roughly the same performance, however.

In general, you should try to use Grand Central Dispatch or operation objects to perform work concurrently. However, there may still be situations where you need to create threads explicitly. For more information about threading support and guidelines on how to use threads safely, see *Threading Programming Guide*.

X11

In Mac OS X v10.3 and later, the X11 windowing system is provided as an optional installation component for the system. This windowing system is used by many UNIX applications to draw windows, controls, and other elements of graphical user interfaces. The Mac OS X implementation of X11 uses the Quartz drawing environment to give X11 windows a native Mac OS X feel. This integration also makes it possible to display X11 windows alongside windows from native applications written in Carbon and Cocoa.

Security

The roots of Mac OS X in the UNIX operating system provide a robust and secure computing environment whose track record extends back many decades. Mac OS X security services are built on top of two open-source standards: BSD (Berkeley Software Distribution) and CDSA (Common Data Security Architecture). BSD is a form of the UNIX operating system that provides basic security for fundamental services, such as file and network access. CDSA provides a much wider array of security services, including finer-grained access permissions, authentication of users' identities, encryption, and secure data storage. Although CDSA has its own standard API, it is complex and does not follow standard Macintosh programming conventions. Therefore, Mac OS X includes its own security APIs that call through to the CDSA API for you.

In Mac OS X v10.5 several improvements were made to the underlying operating system security, including the addition of the following features:

- Adoption of the Mandatory Access Control (MAC) framework, which provides a fine-grained security architecture for controlling the execution of processes at the kernel level. This feature enables the "sandboxing" of applications, which lets you limit the access of a given application to only those features you designate.
- Support for code signing and installer package signing. This feature lets the system validate applications using a digital signature and warn the user if an application is tampered with.
- Compiler support for fortifying your source code against potential security threats. This support includes options to disallow the execution of code located on the stack or other portions of memory containing data. It also includes some new GCC compiler warnings.
- Support for putting unknown files into quarantine. This is especially useful for developers of web browsers or other network-based applications that receive files from unknown sources. The system prevents access to quarantined files unless the user explicitly approves that access.

For an introduction to Mac OS X security features, see *Security Overview*.

Core Technologies

Some technologies are responsible for implementing key portions of your application's infrastructure. The following sections describe these technologies and how use them in your applications.

Blocks

Introduced in Mac OS X v10.6, blocks are a C-level mechanism that you can use to create an ad hoc function body as an inline expression in your code. In other languages and environments, a block is sometimes called a *closure* or a *lambda*. You use blocks when you need to create a reusable segment of code but defining a function or method might be a heavyweight (and perhaps inflexible) solution—for example, if you want to write callbacks with custom data or if you want to perform an operation on all the items in a collection.

The compiler provides support for blocks using the C, C++, and Objective-C languages. For more information about how to create and use blocks, see *Blocks Programming Topics*.

Grand Central Dispatch

Introduced in Mac OS X v10.6, Grand Central Dispatch (GCD) provides a simple and efficient API for achieving the concurrent execution of code in your application. Instead of threads, GCD provides the infrastructure for executing any task in your application asynchronously using a dispatch queue. **Dispatch queues** collect your tasks and work with the kernel to facilitate their execution on an underlying thread. A single dispatch queue can execute tasks serially or concurrently and applications can have multiple dispatch queues executing tasks in parallel.

There are several advantages to using dispatch queues over traditional threads. One of the most important is performance. Dispatch queues work more closely with the kernel to eliminate the normal overhead associated with creating threads. Serial dispatch queues also provide built-in synchronization for queued tasks, eliminating many of the problems normally associated with synchronization and memory contention normally encountered when using threads.

In addition to dispatch queues, GCD provides other interfaces to support the asynchronous design approach offered by dispatch queues:

- Dispatch sources provide a more efficient way to handle the following types of kernel-level events:
  - Timer notifications
  - Signal handling
  - Events associated with file and socket operations
  - Significant process-related events
  - Mach-related events
  - Custom events that you define and trigger
- Dispatch groups allow one thread (or task) to block while it waits for one or more other tasks to finish executing.
- Dispatch semaphores provide a more efficient alternative to the traditional semaphore mechanism.

For more information about how to use GCD in your applications, see *Concurrency Programming Guide*.

OpenCL

Introduced in Mac OS X v10.6, the Open Computing Language (OpenCL) makes the high-performance parallel processing power of GPUs available for general-purpose computing. The OpenCL language is a general purpose computer language, not specifically a graphics language, that abstracts out the lower-level details needed to perform parallel data computation tasks on GPUs and CPUs. Using OpenCL, you create compute kernels that are then offloaded to a graphics card or CPU for processing. Multiple instances of a compute kernel can be run in parallel on one or more GPU or CPU cores, and you can link to your compute kernels from Cocoa, C, or C++ applications.

For tasks that involve data-parallel processing on large data sets, OpenCL can yield significant performance gains. There are many applications that are ideal for acceleration using OpenCL, such as signal processing, image manipulation, or finite element modeling. The OpenCL language has a rich vocabulary of vector and scalar operators and the ability to operate on multidimensional arrays in parallel.

For information about OpenCL and how to write compute kernels, see *OpenCL Programming Guide for Mac OS X*.

Core Foundation

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

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

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

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

IPC and Notification Mechanisms

Mac OS X supports numerous technologies for interprocess communication (IPC) and for delivering notifications across the system. The following sections describe the available technologies.

FSEvents API

Introduced in Mac OS X v10.5, the FSEvents API notifies your application when changes occur in the file system. You can use file system events to monitor directories for any changes, such as the creation, modification, or removal of contained files and directories. Although kqueues provide similar behavior, the FSEvents API provides a much simpler way to monitor many directories at once. For example, you can use file system events to monitor entire file system hierarchies rooted at a specific directory and still receive notifications about individual directories in the hierarchy. The implementation of file system events is lightweight and efficient, providing built-in coalescing when multiple changes occur within a short period of time to one or many directories.

The FSEvents API is not intended for detecting fine-grained changes to individual files. You would not use this to detect changes to an individual file as in a virus checker program. Instead, you might use FSEvents to detect general changes to a file hierarchy. For example, you might use this technology in backup software to detect what files changed. You might also use it to monitor a set of data files your application uses, but which can be modified by other applications as well.

For information on how to use the FSEvents API, see *File System Events Programming Guide*.

Kernel Queues and Kernel Events

Kernel queues (also known as kqueues) and kernel events (also known as kevents) are an extremely powerful technology you use to intercept kernel-level events. Although often used to detect file-system changes, you can also use this technology to receive notifications about changes to sockets, processes, and other aspects of the system. For example, you could use them to detect when a process exits or when it issues `fork` and `exec` calls. Kernel queues and events are part of the FreeBSD layer of the operating system and are described in the `kqueue` and `kevent` man pages.

BSD Notifications

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

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

You can add support for BSD notifications to any type of program, including Carbon and Cocoa applications. For more information, see *Mac OS X Notification Overview* or the `notify` man page.

Sockets, Ports, and Streams

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

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

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

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

BSD Pipes

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

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

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

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

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

Shared Memory

Shared memory is a region of memory that has been allocated by a process specifically for the purpose of being readable and possibly writable among several processes. You create regions of shared memory in several different ways. Among the available options are the functions in `/usr/include/sys/shm.h`, the

`shm_open` and `shm_unlink` routines, and the `mmap` routine. Access to shared memory is controlled through POSIX semaphores, which implement a kind of locking mechanism. Shared memory has some distinct advantages over other forms of interprocess communication:

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

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

For information about `shm_open`, `shm_unlink`, and `mmap`, see the `shm_open`, `shm_unlink`, and `mmap` man pages.

Apple Events

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

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

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

Distributed Notifications

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

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

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

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

Distributed Objects for Cocoa

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

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

Mach Messaging

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

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

Software Development Support

The following sections describe some additional features of Mac OS X that affect the software development process.

Binary File Architecture

The underlying architecture of Mac OS X executables was built from the beginning with flexibility in mind. This flexibility has become important as Macintosh computers have transitioned from using PowerPC to Intel CPUs and from supporting only 32-bit applications to 64-bit applications in Mac OS X v10.5. The following sections provide an overview of the types of architectures you can support in your Mac OS X executables along with other information about the runtime and debugging environments available to you.

Hardware Architectures

When Mac OS X was first introduced, it was built to support a 32-bit PowerPC hardware architecture. With Apple's transition to Intel-based Macintosh computers, Mac OS X added initial support for 32-bit Intel hardware architectures. In addition to 32-bit support, Mac OS X v10.4 added some basic support for 64-bit architectures as well and this support was expanded in Mac OS X v10.5. This means that applications and libraries can now support four different architectures:

- 32-bit Intel (`i386`)
- 32-bit PowerPC (`ppc`)
- 64-bit Intel (`x86_64`)
- 64-bit PowerPC (`ppc64`)

Although applications can support all of these architectures in a single binary, doing so is not required. That does not mean application developers can pick a single architecture and use that alone, however. It is recommended that developers create their applications as "universal binaries" so that they run natively on both 32-bit Intel and PowerPC processors. If performance or development need warrants it, you might also add support for the 64-bit versions of each architecture.

Because libraries can be linked into multiple applications, you might consider supporting all of the available architectures when creating them. Although supporting all architectures is not required, it does give developers using your library more flexibility in how they create their applications and is recommended.

Supporting multiple architectures requires careful planning and testing of your code for each architecture. There are subtle differences from one architecture to the next that can cause problems if not accounted for in your code. For example, the PowerPC and Intel architectures use different endian structures for multi-byte data. In addition, some built-in data types have different sizes in 32-bit and 64-bit architectures. Accounting for these differences is not difficult but requires consideration to avoid coding errors.

Xcode provides integral support for creating applications that support multiple hardware architectures. For information about tools support and creating universal binaries to support both PowerPC and Intel architectures, see *Universal Binary Programming Guidelines, Second Edition*. For information about 64-bit support in Mac OS X, including links to documentation for how to make the transition, see "64-Bit Support" (page 37).

64-Bit Support

Mac OS X was initially designed to support binary files on computers using a 32-bit architecture. In Mac OS X version 10.4, however, support was introduced for compiling, linking, and debugging binaries on a 64-bit architecture. This initial support was limited to code written using C or C++ only. In addition, 64-bit binaries could link against the Accelerate framework and `libSystem.dylib` only.

In Mac OS X v10.5, most system libraries and frameworks are now 64-bit ready, meaning they can be used in both 32-bit and 64-bit applications. The conversion of frameworks to support 64-bit required some implementation changes to ensure the proper handling of 64-bit data structures; however, most of these changes should be transparent to your use of the frameworks. Building for 64-bit means you can create applications that address extremely large data sets, up to 128TB on the current Intel-based CPUs. On Intel-based Macintosh computers, some 64-bit applications may even run faster than their 32-bit equivalents because of the availability of extra processor resources in 64-bit mode.

Although most APIs support 64-bit development, some older APIs were not ported to 64-bit or offer restricted support for 64-bit applications. Many of these APIs are legacy Carbon managers that have been either wholly or partially deprecated in favor of more modern equivalents. What follows is a partial list of APIs that will not support 64-bit. For a complete description of 64-bit support in Carbon, see *64-Bit Guide for Carbon Developers*.

- Code Fragment Manager (use the Mach-O executable format instead)
- Desktop Manager (use Icon Services and Launch Services instead)
- Display Manager (use Quartz Services instead)
- QuickDraw (use Quartz or Cocoa instead)
- QuickTime Musical Instruments (use Core Audio instead)
- Sound Manager (use Core Audio instead)

In addition to the list of deprecated APIs, there are a few modern APIs that are not deprecated, but which have not been ported to 64-bit. Development of 32-bit applications with these APIs is still supported, but if you want to create a 64-bit application, you must use alternative technologies. Among these APIs are the following:

- The entire QuickTime C API (not deprecated, but developers should use QuickTime Kit instead in 64-bit applications)
- HIToolbox, Window Manager, and most other Carbon user interface APIs (not deprecated, but developers should use Cocoa user interface classes and other alternatives); see *64-Bit Guide for Carbon Developers* for the list of specific APIs and transition paths.

Mac OS X uses the LP64 model that is in use by other 64-bit UNIX systems, which means fewer headaches when porting from other operating systems. For general information on the LP64 model and how to write 64-bit applications, see *64-Bit Transition Guide*. For Cocoa-specific transition information, see *64-Bit Transition Guide for Cocoa*. For Carbon-specific transition information, see *64-Bit Guide for Carbon Developers*.

Object File Formats

Mac OS X is capable of loading object files that use several different object-file formats, including the following:

- Mach-O
- Java bytecode
- Preferred Executable Format (PEF)

Of these formats, the Mach-O format is the format used for all native Mac OS X application development. The Java bytecode format is a format executed through the Hotspot Java virtual machine and used exclusively for Java-based programs. The PEF format is handled by the Code Fragment Manager and is a legacy format that was used for transitioning Mac OS 9 applications to Mac OS X.

For information about the Mach-O file format, see *Mac OS X ABI Mach-O File Format Reference*. For additional information about using Mach-O files, see *Mach-O Programming Topics*. For information about Java support in Mac OS X, see "Java Support" (page 41). For information about the PEF format and Code Fragment Manager, see "CFM Runtime Environment" (page 40)

Debug File Formats

Whenever you debug an executable file, the debugger uses symbol information generated by the compiler to associate user-readable names with the procedure and data address it finds in memory. Normally, this user-readable information is not needed by a running program and is stripped out (or never generated) by the compiler to save space in the resulting binary file. For debugging, however, this information is very important to be able to understand what the program is doing.

Mac OS X supports two different debug file formats for compiled executables: stabs and DWARF. The stabs format is present in all versions of Mac OS X and until the introduction of Xcode 2.4 was the default debugging format. Code compiled with Xcode 2.4 and later uses the DWARF debugging format by default. When using the stabs format, debugging symbols, like other symbols are stored in the symbol table of the executable; see *Mac OS X ABI Mach-O File Format Reference*. With the DWARF format, however, debugging symbols are stored either in a specialized segment of the executable or in a separate debug-information file.

For information about the DWARF standard, go to http://www.dwarfstd.org. For information about the stabs debug file format, see *STABS Debug Format*. For additional information about Mach-O files and their stored symbols, see *Mach-O Programming Topics*.

Runtime Environments

Since its first release, Mac OS X has supported several different environments for running applications. The most prominent of these environments is the Dyld environment, which is also the only environment supported for active development. Most of the other environments provided legacy support during the transition from Mac OS 9 to Mac OS X and are no longer supported for active development. The following sections describe the runtime environments you may encounter in various versions of Mac OS X.

Dyld Runtime Environment

The dyld runtime environment is the native environment in Mac OS X and is used to load, link, and execute Mach-O files. At the heart of this environment is the `dyld` dynamic loader program, which handles the loading of a program's code modules and associated dynamic libraries, resolves any dependencies between those libraries and modules, and begins the execution of the program.

Upon loading a program's code modules, the dynamic loader performs the minimal amount of symbol binding needed to launch your program and get it running. This binding process involves resolving links to external libraries and loading them as their symbols are used. The dynamic loader takes a lazy approach to binding individual symbols, doing so only as they are used by your code. Symbols in your code can be strongly-linked or weakly-linked. Strongly-linked symbols cause the dynamic loader to terminate your program if the library containing the symbol cannot be found or the symbol is not present in the library. Weakly-linked symbols terminate your program only if the symbol is not present and an attempt is made to use it.

For more information about the dynamic loader program, see `dyld`. For information about building and working with Mach-O executable files, see *Mach-O Programming Topics*.

Java Runtime Environment

The Java runtime environment consists of the HotSpot Java virtual machine, the "just-in-time" (JIT) bytecode compiler, and code packages containing the standard Java classes. For more information about Java support in Mac OS X, see "Java Support" (page 41).

CFM Runtime Environment

The Code Fragment Manager (CFM) runtime environment is a legacy environment inherited from Mac OS 9. Mac OS X provides this environment to support applications that want to use the modern features of Mac OS X but have not yet been converted over to the dyld environment for various reasons. The CFM runtime environment expects code modules to be built using the Preferred Executable Format (PEF).

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

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

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

On Intel-based Macintosh computers, CFM binaries are run under the Rosetta environment.

The Classic Environment

Important: The Classic environment was supported only on PowerPC-based Macintosh computers and was deprecated in Mac OS X v10.5 and later. You should not be doing any active development using the Classic environment. If you want to write programs to run in Mac OS X, you should use the dyld environment instead.

In early versions of Mac OS X, the Classic compatibility environment (or simply, Classic environment) was called a "software compatibility" environment because it enabled Mac OS X to run applications built for Mac OS 9.1 or 9.2. The Classic environment was not an emulator; it was a hardware abstraction layer between an installed Mac OS 9 System Folder and the Mac OS X kernel environment. Because of architectural differences, applications running in the Classic environment did not share the full advantages of the kernel environment.

The Classic environment is not supported on Intel-based Macintosh computers.

Language Support

The tools that come with Mac OS X provide direct support for developing software using the C, C++, Objective-C, Objective-C++, languages along with numerous scripting languages. It also includes a Java runtime that you can use to write Java applications. Support for other languages may also be provided by third-party developers.

The following sections call out key features in some of these environments.

Objective-C

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

In Mac OS X v10.5, an update to the Objective-C language (called Objective-C 2.0) was introduced, adding support for the following features:

- Object properties, which offer an alternative way to declare member variables
- Support for garbage collection; see *Garbage Collection Programming Guide*
- A new `for` operator syntax for performing fast enumerations of collections
- Protocol enhancements
- Deprecation syntax

In Mac OS X v10.6, support was added for blocks, which are described in "Blocks" (page 31).

For information about the Objective-C language, see *The Objective-C Programming Language*.

Java Support

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

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

The Java Environment

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

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

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

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

Java and Other Application Environments

Java applications can take advantage of Mac OS X technologies such as Cocoa and QuickTime through Sun's Java Native Interface (JNI). For details on using the JNI on Mac OS X, see *Technical Note 2147*.

Graphics and Multimedia Technologies

The graphics and multimedia capabilities of Mac OS X set it apart from other operating systems. Mac OS X is built on a modern foundation that includes support for advanced compositing operations with support for hardware-based rendering on supported graphics hardware. On top of this core are an array of technologies that provide support for drawing 2D, 3D, and video-based content. The system also provides an advanced audio system for the generation, playback, and manipulation of multichannel audio.

Drawing Technologies

Mac OS X includes numerous technologies for rendering 2D and 3D content and for animating that content dynamically at runtime.

Quartz

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

Quartz comprises both a client API (Quartz 2D) and a window server (Quartz Compositor). The client API provides commands for managing the graphics context and for drawing primitive shapes, images, text, and other content. The window server manages the display and device driver environment and provides essential services to clients, including basic window management, event routing, and cursor management behaviors.

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

The Quartz Services API (which is also part of the Core Graphics framework) provides direct access to some low-level features of the window server. You can use this API to get information about the currently connected display hardware, capture a display for exclusive use, or adjust display attributes, such as its resolution, pixel depth, and refresh rate. Quartz Services also provides some support for operating a Mac OS X system remotely.

For information about the Quartz 2D API, see *Quartz 2D Programming Guide*. For information about the Quartz Services API, see *Quartz Display Services Programming Topics*.

Digital Paper Metaphor

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

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

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

Quartz 2D Features

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

- High-quality rendering on the screen
- Resolution independent UI support
- Anti-aliasing for all graphics and text
- Support for adding transparency information to windows
- Internal compression of data
- A consistent feature set for all printers
- Automatic PDF generation and support for printing, faxing, and saving as PDF
- Color management through ColorSync

Table 3-1 describes some of technical specifications for Quartz.

Table 3-1 Quartz technical specifications

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

Quartz Compositor

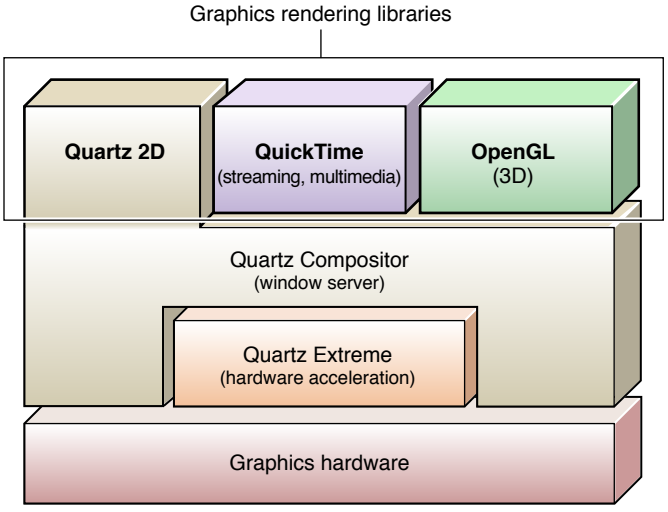
Quartz Compositor, the window server for Mac OS X, coordinates all of the low-level windowing behavior and enforces a fundamental uniformity in what appears on the screen. It manages the displays available on the user's system, interacting with the necessary device drivers. It also provides window management,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

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

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

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

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



Cocoa Drawing

The Cocoa application environment provides object-oriented wrappers for many of the features found in Quartz. Cocoa provides support for drawing primitive shapes such as lines, rectangles, ovals, arcs, and Bezier paths. It supports drawing in both standard and custom color spaces and it supports content manipulations using graphics transforms. Because it is built on top of Quartz, drawing calls made from Cocoa are composited along with all other Quartz 2D content. You can even mix Quartz drawing calls (and drawing calls from other system graphics technologies) with Cocoa calls in your code if you wish.

For more information on how to draw using Cocoa, see *Cocoa Drawing Guide*.

OpenGL

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

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

In Mac OS X v10.5 and later, OpenGL supports the ability to use multiple threads to process graphics data. OpenGL also supports pixel buffer objects, color managed texture images in the sRGB color space, support for 64-bit addressing, and improvements in the shader programming API. You can also attach an AGL context to `WindowRef` and `HIView` objects and thereby avoid using QuickDraw ports.

For information about using OpenGL in Mac OS X, see *OpenGL Programming Guide for Mac OS X*.

Core Animation

Introduced in Mac OS X v10.5, Core Animation is a set of Objective-C classes for doing sophisticated 2D rendering and animation. Using Core Animation, you can create everything from basic window content to Front Row–style user interfaces, and achieve respectable animation performance, without having to tune your code using OpenGL or other low-level drawing routines. This performance is achieved using server-side content caching, which restricts the compositing operations performed by the server to only those parts of a view or window whose contents actually changed.

At the heart of the Core Animation programming model are layer objects, which are similar in many ways to Cocoa views. Like views, you can arrange layers in hierarchies, change their size and position, and tell them to draw themselves. Unlike views, layers do not support event-handling, accessibility, or drag and drop. You can also manipulate the layout of layers in more ways than traditional Cocoa views. In addition to positioning layers using a layout manager, you can apply 3D transforms to layers to rotate, scale, skew, or translate them in relation to their parent layer.

Layer content can be animated implicitly or explicitly depending on the actions you take. Modifying specific properties of a layer, such as its geometry, visual attributes, or children, typically triggers an implicit animation to transition from the old state to the new state of the property. For example, adding a child layer triggers an animation that causes the child layer to fade gradually into view. You can also trigger animations explicitly in a layer by modifying its transformation matrix.

You can manipulate layers independent of, or in conjunction with, the views and windows of your application. Both Cocoa and Carbon applications can take advantage of the Core Animation's integration with the `NSView` class to add animation effects to windows. Layers can also support the following types of content:

- Quartz Composer compositions
- OpenGL content
- Core Image filter effects
- Quartz and Cocoa drawing content
- QuickTime playback and capture

The Core Animation features are part of the Quartz Core framework (`QuartzCore.framework`). For information about Core Animation, see *Animation Overview*.

Core Image

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

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

- Crop images
- Correct color, including perform white-point adjustments
- Apply color effects, such as sepia tone
- Blur or sharpen images
- Composite images
- Warp the geometry of an image by applying an affine transform or a displacement effect
- Generate color, checkerboard patterns, Gaussian gradients, and other pattern images
- Add transition effects to images or video
- Provide real-time control, such as color adjustment and support for sports, vivid, and other video modes
- Apply linear lighting effects, such as spotlight effects

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

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

Image Kit

Introduced in Mac OS X v10.5, the Image Kit framework is an Objective-C framework that makes it easy to incorporate powerful imaging services into your applications. This framework takes advantage of features in Quartz, Core Image, OpenGL, and Core Animation to provide an advanced and highly optimized development path for implementing the following features:

- Displaying images
- Rotating, cropping, and performing other image-editing operations
- Browsing for images
- Taking pictures using the built-in picture taker panel
- Displaying slideshows
- Browsing for Core Image filters
- Displaying custom views for Core Image filters

The Image Kit framework is included as a subframework of the Quartz framework (`Quartz.framework`). For more information on how to use Image Kit, see *Image Kit Programming Guide* and *Image Kit Reference Collection*

QuickDraw

QuickDraw is a legacy technology adapted from earlier versions of the Mac OS that lets you construct, manipulate, and display two-dimensional shapes, pictures, and text. Because it is a legacy technology, QuickDraw should not be used for any active development. Instead, you should use Quartz.

If your code currently uses QuickDraw, you should begin converting it to Quartz 2D as soon as possible. The QuickDraw API includes features to make transitioning your code easier. For example, QuickDraw includes interfaces for getting a Quartz graphics context from a `GrafPort` structure. You can use these interfaces to transition your QuickDraw code in stages without radically impacting the stability of your builds.

Important: QuickDraw is deprecated in Mac OS X v10.5 and later. QuickDraw is not available for 64-bit applications.

Text and Fonts

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

Cocoa Text

Cocoa provides advanced text-handling capabilities in the Application Kit framework. Based on Core Text, the Cocoa text system provides a multilayered approach to implementing a full-featured text system using Objective-C. This layered approach lets you customize portions of the system that are relevant to your needs while using the default behavior for the rest of the system. You can use Cocoa Text to display small or large amounts of text and can customize the default layout manager classes to support custom layout.

Although part of Cocoa, the Cocoa text system can also be used in Carbon-based applications. If your Carbon application displays moderate amounts of read-only or editable text, you can use `HIView` wrappers for the `NSString`, `NSTextField`, and `NSTextView` classes to implement that support. Using wrappers is much easier than trying to implement the same behavior using lower-level APIs, such as Core Text, ATSUI, or MLTE. For more information on using wrapper classes, see *Carbon-Cocoa Integration Guide*.

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

Core Text

Introduced in Mac OS X v10.5, Core Text is a C-based API that provides you with precise control over text layout and typography. Core Text provides a layered approach to laying out and displaying Unicode text. You can modify as much or as little of the system as is required to suit your needs. Core Text also provides optimized configurations for common scenarios, saving setup time in your application. Designed for performance, Core Text is up to twice as fast as ATSUI (see "Apple Type Services for Unicode Imaging" (page 50)), the text-handling technology that it replaces.

The Core Text font API is complementary to the Core Text layout engine. Core Text font technology is designed to handle Unicode fonts natively and comprehensively, unifying disparate Mac OS X font facilities so that developers can do everything they need to do without resorting to other APIs.

Carbon and Cocoa developers who want a high-level text layout API should consider using the Cocoa text system and the supporting Cocoa text views. Unless you need low-level access to the layout manager routines, the Cocoa text system should provide most of the features and performance you need. If you need a lower-level API for drawing any kind of text into a `CGContext`, then you should consider using the Core Text API.

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

Apple Type Services

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

Note: In Mac OS X v10.5 and later, you should consider using the Core Text font-handling API instead of this technology. For more information, see "Core Text" (page 49).

Apple Type Services for Unicode Imaging

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

Note: In Mac OS X v10.5 and later, you should use the Core Text API instead of this technology. For more information, see "Core Text" (page 49).

Multilingual Text Engine

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

In Mac OS X v10.5 and later, the QuickDraw-related features of MLTE are deprecated. The features that use HITextView are still supported, however.

Note: In Mac OS X v10.5 and later, you should use the Core Text API instead of this technology. For more information, see "Core Text" (page 49).

Audio Technologies

Mac OS X includes support for high-quality audio creation and reproduction.

Core Audio

The Core Audio frameworks of Mac OS X offer a sophisticated set of services for manipulating multichannel audio. You can use Core Audio to generate, record, mix, edit, process, and play audio. You can also use Core Audio to generate, record, process, and play MIDI data using both hardware and software MIDI instruments.

For the most part, the interfaces of the Core Audio frameworks are C-based, although some of the Cocoa-related interfaces are implemented in Objective-C. The use of C-based interfaces results in a low-latency, flexible programming environment that you can use from both Carbon and Cocoa applications. Some of the benefits of Core Audio include the following:

- Built-in support for reading and writing a wide variety of audio file and data formats
- Plug-in interfaces for handling custom file and data formats
- Plug-in interfaces for performing audio synthesis and audio digital signal processing (DSP)
- A modular approach for constructing audio signal chains
- Scalable multichannel input and output
- Easy synchronization of audio MIDI data during recording or playback
- Support for playing and recording digital audio, including support for scheduled playback and synchronization and for getting timing and control information
- A standardized interface to all built-in and external hardware devices, regardless of connection type (USB, Firewire, PCI, and so on)

For an overview of Core Audio and its features, see *Core Audio Overview*. For reference information, see *Core Audio Framework Reference*.

OpenAL

Introduced in Mac OS X v10.4, the Open Audio Library (OpenAL) audio system adds another way to create audio for your software. The OpenAL interface is a cross-platform standard for delivering 3D audio in applications. It lets you implement high-performance positional audio in games and other programs that require high-quality audio output. Because it is a cross-platform standard, the applications you write using OpenAL on Mac OS X can be ported to run on many other platforms.

In Mac OS X v10.5, several features were incorporated into the existing OpenAL framework. Among these features are support for audio capture, exponential and linear distance models, location offsets, and spatial effects such as reverb and occlusion. In addition, more control is provided for some Core Audio features such as mixer sample rates.

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

For more information on the Mac OS X implementation of OpenAL, go to http://developer.apple.com/audio/openal.html.

Video Technologies

The video technologies in Mac OS X allow you to work with movies and other time-based content, including audio.

QuickTime Kit

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

In Mac OS X v10.5, support was added for capturing professional-quality audio and video content from one or more external sources, including cameras, microphones, USB and Firewire devices, DV media devices, QuickTime streams, data files, and the screen. The input and output classes included with the framework provide all of the components necessary to implement the most common use case for a media capture system: recording from a camera to a QuickTime file. Video capture includes frame accurate audio/video synchronization, plus you can preview captured content and save it to a file or stream.

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

In Mac OS X v10.6, the QT Kit framework takes advantage of the new QuickTime X media services to allow applications to open movies asynchronously on supported media types. The framework also uses these media services to manage most of the interactions with media files.

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

Core Video

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

In Mac OS X v10.4, Core Video is part of the Quartz Core framework (`QuartzCore.framework`). In Mac OS X v10.5 and later, the interfaces are duplicated in the Core Video framework (`CoreVideo.framework`).

For information about using the Core Video framework, see *Core Video Programming Guide*.

DVD Playback

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

For more information about using the DVD Playback framework, see*DVD Playback Services Programming Guide*.

QuickTime

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

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

For a long time, QuickTime has included programming interfaces for the C and C++ languages. Beginning with Mac OS X v10.4, the QuickTime Kit provides an Objective-C based set of classes for managing QuickTime content. For more information about QuickTime Kit, see "QuickTime Kit" (page 52).

Note: In Mac OS X v10.5 and later, you must use the QuickTime Kit framework to create 64-bit applications. The QuickTime C-based APIs are not supported in 64-bit applications.

Supported Media Formats

QuickTime supports more than a hundred media types, covering a range of audio, video, image, and streaming formats. Table 3-2 lists some of the more common file formats it supports. For a complete list of supported formats, see the QuickTime product specification page at http://www.apple.com/quicktime/pro/specs.html.

Table 3-2 Partial list of formats supported by QuickTime

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

Extending QuickTime

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

You can extend QuickTime by writing your own component. You might write your own QuickTime component to support a new media type or to implement a new codec. You might also write components to support a custom video capture card. By implementing your code as a QuickTime component that you enable, other applications take advantage of your code and use it to support your hardware or media file formats. See "QuickTime Components" (page 83) for more information.

Color Management

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

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

In Mac OS X 10.6 and later, the default system gamma value switched from 1.8 (a useful value for print professionals) to 2.2, which is used more prevalently in television, video, and the web. Images that are already tagged with gamma information will look the same as they did in previous versions of Mac OS X. To prevent the unwanted darkening of untagged user interface elements such as icons, Mac OS X v10.6 automatically adjusts those images to account for the change in gamma. Thus, applications that use system UI elements such as Cocoa controls will see little or no change. However, you still need to adjust your application's custom artwork to account for the gamma change.

For information about the ColorSync API, see *ColorSync Manager Reference*. For additional information about the gamma changes in Mac OS X v10.6, see Mac OS X v10.6 in *What's New In Mac OS X*.

Printing

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

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

Feature	Description
CUPS	The Common Unix Printing System (CUPS) provides the underlying support for printing. It is an open-source architecture used commonly by the UNIX community to handle print spooling and other low-level features.
Desktop printers	In Mac OS X v10.3 and later, the system supports desktop printers, which offer users a way to manage printing from the Dock or desktop. Users can print natively supported files (like PostScript and PDF) by dragging them to a desktop printer. Users can also manage print jobs.
Fax support	In Mac OS X v10.3 and later, users can fax documents directly from the Print dialog.
GIMP-Print drivers	In Mac OS X v10.3 and later, the system includes drivers for many older printers through the print facility of the GNU Image Manipulation Program (GIMP).
Native PDF	Supports PDF as a native data type. Any application (except for Classic applications) can easily save textual and graphical data to device-independent PDF where appropriate. The printing system provides this capability from a standard printing dialog.
PostScript support	Mac OS X prints to PostScript Level 2–compatible and Level 3–compatible printers. In Mac OS X v10.3 and later, support is also provided to convert PostScript files directly to PDF.
Print preview	Provides a print preview capability in all environments, except in Classic. The printing system implements this feature by launching a PDF viewer application. This preview is color-managed by ColorSync.
Printer discovery	Printers implementing Bluetooth or Bonjour can be detected, configured, and added to printer lists automatically.
Raster printers	Supports printing to raster printers in all environments, except in the Classic environment.
Speedy spooling	In Mac OS X v10.3 and later, applications that use PDF can submit PDF files directly to the printing system instead of spooling individual pages. This simplifies printing for applications that already store data as PDF.

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

Accelerating Your Multimedia Operations

Mac OS X takes advantage of hardware wherever it can to improve performance wherever it can. In the case of repetitive tasks operating on large data sets, Mac OS X uses the vector-oriented extensions provided by the processor. (Mac OS X currently supports the PowerPC AltiVec extensions and the Intel x86 SSE extensions.) Hardware-based vector units boost the performance of any application that exploits data parallelism, such

as those that perform 3D graphic imaging, image processing, video processing, audio compression, and software-based cell telephony. Quartz and QuickTime incorporate vector capabilities, thus any application using these APIs can tap into this hardware acceleration without making any changes.

In Mac OS X v10.3 and later, you can use the Accelerate framework (`Accelerate.framework`) to accelerate complex operations using the available vector unit. This framework supports both the PowerPC AltiVec and Intel x86 SSE extensions internally but provides a single interface for you to use in your application. The advantage of using this framework is that you can simply write your code once without having to code different execution paths for each hardware platform. The functions of this framework are highly tuned for the specific platforms supported by Mac OS X and in many cases can offer better performance than hand-rolled code.

The Accelerate framework is an umbrella framework that wraps the vecLib and vImage frameworks into a single package. The vecLib framework contains vector-optimized routines for doing digital signal processing, linear algebra, and other computationally expensive mathematical operations. (The vecLib framework is also a top-level framework for applications running on versions of Mac OS X up to and including version 10.5.) The vImage framework supports the visual realm, adding routines for morphing, alpha-channel processing, and other image-buffer manipulations.

For information on how to use the components of the Accelerate framework, see *vImage Programming Guide*, *vImage Reference Collection*, and *vecLib Reference*. For general performance-related information, see Reference Library > Performance.

Application Technologies

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

Application Environments

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

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

Cocoa

Cocoa is an object-oriented environment designed for rapid application development. It features a sophisticated framework of objects for implementing your application and takes full advantage of graphical tools such as Interface Builder to enable you to create full-featured applications quickly and without a lot of code. The Cocoa environment is especially suited for:

- New developers
- Developers who prefer working with object-oriented systems
- Developers who need to prototype an application quickly
- Developers who prefer to leverage the default behavior provided by the Cocoa frameworks so they can focus on the features unique to their application
- Objective-C or Objective-C++ developers
- Python and Ruby developers who want to take advantage of Cocoa features; see *Ruby and Python Programming Topics for Mac OS X*

The objects in the Cocoa framework handle much of the behavior required of a well-behaved Mac OS X application, including menu management, window management, document management, Open and Save dialogs, and pasteboard (clipboard) behavior. Cocoa's support for Interface Builder means that you can create most of your user interface (including much of its behavior) graphically rather than programatically. With the addition of Cocoa bindings and Core Data, you can also implement most of the rest of your application graphically as well.

The Cocoa application environment consists of two object-oriented frameworks: Foundation (`Foundation.framework`) and the Application Kit (`AppKit.framework`). The classes in the Foundation framework implement data management, file access, process notification, memory management, network communication, and other low-level features. The classes in the Application Kit framework implement the user interface layer of an application, including windows, dialogs, controls, menus, and event handling. If you are writing an application, link with the Cocoa framework (`Cocoa.framework`), which imports both the Foundation and Application Kit frameworks. If you are writing a Cocoa program that does not have a graphical user interface (a background server, for example), you can link your program solely with the Foundation framework.

Apple's developer documentation contains a section devoted to Cocoa where you can find conceptual material, reference documentation, and tutorials showing how to write Cocoa applications. If you are a new Cocoa developer, be sure to read *Cocoa Fundamentals Guide*, which provides an in-depth overview of the development process for Cocoa applications. For information about the development tools, including Interface Builder, see "Mac OS X Developer Tools" (page 127).

Carbon

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

The Carbon environment is especially suited for:

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

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

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

In Mac OS X v10.5 and later, Carbon includes support for integrating Cocoa views into your Carbon applications. After creating the Cocoa view, you can wrap it in an HIView object and embed that object in your window. Once embedded, you use the standard HIView functions to manipulate the view. Wrapped Cocoa views can be used in both composited and noncomposited windows to support views and controls that are available in Cocoa but are not yet available in Carbon. For more information, see *Carbon-Cocoa Integration Guide* and *HIView Reference*.

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

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

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

Java

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

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

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

Note: Any Mach-O binaries that interact with the JVM must be universal binaries. This includes JNI libraries as well as traditional applications that invoke the JVM. For more information, see *Universal Binary Programming Guidelines, Second Edition*.

For details on the tools and support provided for Java developers, see "Java Support" (page 41).

WebObjects

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

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

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

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

BSD and X11

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

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

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

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

The BSD environment is part of the Darwin layer of Mac OS X. For information about Darwin, see Reference Library > Darwin. For more information about X11 development, see http://developer.apple.com/darwin/projects/X11. See also "Information on BSD" (page 15) for links to additional BSD resources.

Application Technologies

Mac OS X includes several technologies that make developing applications easier. These technologies range from utilities for managing your internal data structures to high-level frameworks for burning CDs and DVDs. This section summarizes the application-level technologies that are relevant to developers—that is, that have programmatic interfaces or have an impact on how you write software. It does not describe user-level technologies, such as Exposé, unless there is some aspect of the technology that allows developer involvement.

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

Address Book Framework

Introduced in Mac OS X v10.2, Address Book is technology that encompasses a centralized database for contact and group information, an application for viewing that information, and a programmatic interface for accessing that information in your own programs. The database contains information such as user names, street addresses, email addresses, phone numbers, and distribution lists. Applications that support this type of information can use this data as is or extend it to include application-specific information.

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

For more information about this technology, see *Address Book Programming Guide for Mac OS X* and either *Address Book Objective-C Framework Reference for Mac OS X* or *Address Book C Framework Reference for Mac OS X*.

Automator Framework

Introduced in Mac OS X v10.5, the Automator framework (`Automator.framework`) adds support for running workflows from your applications. Workflows are products of the Automator application; they string together the actions defined by various applications to perform complex tasks automatically. Unlike AppleScript, which uses a scripting language to implement the same behavior, workflows are constructed visually, requiring no coding or scripting skills to create.

For information about incorporating workflows into your own applications, see *Automator Framework Reference*.

Bonjour

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

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

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

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

Calendar Store Framework

Introduced in Mac OS X v10.5, the Calendar Store framework (`CalendarStore.framework`) lets you access iCal data from an Objective-C based application. You can use this framework to fetch user calendars, events, and tasks from the iCal data storage, receive notifications when those objects change, and make changes to the user's calendar.

For information about using the Calendar Store framework, see [Calendar Store Programming Guide] and *Calendar Store Programming Guide*.

Core Data Framework

Introduced in Mac OS X version 10.4, the Core Data framework (`CoreData.framework`) manages the data model of a Cocoa-based Model-View-Controller application. Core Data is intended for use in applications where the data model is already highly structured. Instead of defining data structures programmatically, you use the graphical tools in Xcode to build a schema representing your data model. At runtime, instances of your data-model entities are created, managed, and made available through the Core Data framework with little or no coding on your part.

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

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

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

Disc Recording Framework

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

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

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

For more information, see *Disc Recording Framework Reference* and *Disc Recording UI Framework Reference*.

Help Support

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

In Mac OS X v10.5 and later, the Spotlight for Help feature makes it easier for users to locate items in complex menu bars. For applications with a Help menu, Mac OS X automatically inserts a special search field menu item at the top of the menu. When the user enters a string in this search field, the system searches the application menus for commands containing the specified string. Moving the mouse over a search result reveals the location of that item in the menus. Developers do not need to add any code to their applications to support this feature.

For information on adding help to a Cocoa application, see *Online Help*. For information on adding help to a Carbon application, see *Apple Help Programming Guide*.

Human Interface Toolbox

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

- Drawing is handled natively using Quartz.
- A simplified, modern coordinate system is used that is not bounded by the 16-bit space of QuickDraw.
- Support for arbitrary views is provided.
- Layering of views is handled automatically.
- Views can be attached and detached from windows.
- Views can be hidden temporarily.
- You can use Interface Builder to create your interfaces.

Note: The HIToolbox interfaces are available for creating 32-bit applications only. If you are creating 64-bit applications, you should use Cocoa for your user interface instead.

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

Identity Services

Introduced in Mac OS X v10.5, Identity Services encompasses features located in the Collaboration and Core Services frameworks. Identity Services provides a way to manage groups of users on a local system. In addition to standard login accounts, administrative users can now create **sharing** accounts, which use access control lists to restrict access to designated system or application resources. Sharing accounts do not have an associated home directory on the system and have much more limited privileges than traditional login accounts.

The Collaboration framework (`Collaboration.framework`) provides a set of Objective-C classes for displaying sharing account information and other identity-related user interfaces. The classes themselves are wrappers for the C-based identity management routines found in the Core Services framework. Applications can use either the Objective-C or C-based APIs to display information about users and groups and display a panel for selecting users and groups during the editing of access control lists.

For more information about the features of Identity Services and how you use those features in your applications, see *Identity Services Programming Guide* and *Identity Services Reference Collection*.

Instant Message Framework

Introduced in Mac OS X version 10.4, the Instant Message framework (`InstantMessage.framework`) supports the detection and display of a user's online presence in applications other than iChat. You can find out the current status of a user connected to an instant messaging service, obtain the user's custom icon and status message, or obtain a URL to a custom image that indicates the user's status. You can use this information to display the user's status in your own application. For example, Mail identifies users who are currently online by tagging that user's email address with a special icon.

In Mac OS X v10.5, you can use the Instant Message framework to support iChat Theater. This feature gives your application the ability to inject audio or video content into a running iChat conference. The content you provide is then mixed with the user's live microphone and encoded automatically into the H.264 video format for distribution to conference attendees.

For more information about using the Instant Message framework, see *Instant Message Programming Guide*.

Image Capture Services

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

This framework works in conjunction with the Image Capture Devices framework (`ICADevices.framework`) to communicate with imaging hardware. For information on how to use the Image Capture Services framework, see *Image Capture Applications Programming Guide*.

Ink Services

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

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

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

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

Input Method Kit Framework

Introduced in Mac OS X v10.5, the Input Method Kit (`InputMethodKit.framework`) is an Objective-C framework for building input methods for Chinese, Japanese, and other languages. The Input Method Kit framework lets developers focus exclusively on the development of their input method product's core behavior: the text conversion engine. The framework handles tasks such as connecting to clients, running candidate windows, and several other common tasks that developers would normally have to implement themselves.

For information about its classes, see *Input Method Kit Framework Reference*.

Keychain Services

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

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

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

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

Latent Semantic Mapping Services

Introduced in Mac OS X v10.5, the Latent Semantic Mapping framework (`LatentSemanticMapping.framework`) contains a Unicode-based API that supports the classification of text and other token-based content into developer-defined categories, based on semantic information latent in the text. Using this API and text samples with known characteristics, you create and train maps, which you can use to analyze and classify arbitrary text. You might use such a map to determine, for example, if an email message is consistent with the user's interests.

For information about the Latent Semantic Mapping framework, see *Latent Semantic Mapping Reference*.

Launch Services

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

For information on how to use Launch Services, see *Launch Services Programming Guide*.

Open Directory

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

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

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

PDF Kit Framework

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

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

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

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

Publication Subscription Framework

Introduced in Mac OS X v10.5, the Publication Subscription framework (`PubSub.framework`) is a new framework that provides high-level support for subscribing to RSS and Atom feeds. You can use the framework to subscribe to podcasts, photocasts, and any other feed-based document. The framework handles all the feed downloads and updates automatically and provides your application with the data from the feed.

For information about the Publication Subscription framework, see *Publication Subscription Programming Guide* and *Publication Subscription Framework Reference*.

Search Kit Framework

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

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

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

Security Services

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

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

- User authentication

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

Mac OS X supports many network-based security standards, including SFTP, S/MIME, and SSH. For a complete list of network protocols, see "Standard Network Protocols" (page 25).

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

Speech Technologies

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

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

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

For more information, see Reference Library > User Experience > Speech Technologies.

SQLite Library

Introduced in Mac OS X version 10.4, the SQLite library lets you embed a SQL database engine into your applications. Programs that link with the SQLite library can access SQL databases without running a separate RDBMS process. You can create local database files and manage the tables and records in those files. The library is designed for general purpose use but is still optimized to provide fast access to database records.

The SQLite library is located at `/usr/lib/libsqlite3.dylib` and the `sqlite3.h` header file is in `/usr/include`. A command-line interface (`sqlite3`) is also available for communicating with SQLite databases using scripts. For details on how to use this command-line interface, see `sqlite3` man page.

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

Sync Services Framework

Introduced in Mac OS X version 10.4, the Sync Services framework gives you access to the data synchronization engine built-in to Mac OS X. You can use this framework to synchronize your application data with system databases, such as those provided by Address Book and iCal. You can also publish your application's custom data types and make them available for syncing. You might do this to share your application's data with other applications on the same computer or with applications on multiple computers (through the user's .Mac account).

With the Sync Services framework, applications can directly initiate the synchronization process. Prior to Mac OS X v10.4, synchronization occurred only through the iSync application. In Mac OS X v10.4 and later, the iSync application still exists but is used to initiate the synchronization process for specific hardware devices, like cell phones.

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

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

WebKit Framework

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

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

Also introduced in Mac OS X version 10.4, WebKit includes support for creating and editing content at the DOM level of an HTML document. You can use this support to navigate DOM nodes and manipulate those nodes and their attributes. You can also use the framework to extract DOM information. For example, you could extract the list of links on a page, modify them, and replace them prior to displaying the document in a web view.

For information on how to use WebKit from both Carbon and Cocoa applications, see *WebKit Objective-C Programming Guide*. For information on the classes and protocols in the WebKit framework, see *WebKit Objective-C Framework Reference*.

Time Machine Support

Introduced in Mac OS X v10.5, the Time Machine feature protects user data from accidental loss by automatically backing up data to a different hard drive. Included with this feature is a set of programmer-level functions that you can use to exclude unimportant files from the backup set. For example, you might use these functions to exclude your application's cache files or any files that can be recreated easily. Excluding these types of files improves backup performance and reduces the amount of space required to back up the user's system.

For information about the new functions, see *Backup Core Reference*.

Web Service Access

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

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

XML Parsing Libraries

In Mac OS X v10.3, the Darwin layer began including the `libXML2` library for parsing XML data. This is an open source library that you can use to parse or write arbitrary XML data quickly. The headers for this library are located in the `/usr/include/libxml2` directory.

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

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

User Experience

One reason users choose the Macintosh over other platforms is that it provides a compelling user experience. This user experience is defined partly by the technologies and applications that are built-in to Mac OS X and partly by the applications you create. Your applications play a key role in delivering the experience users expect. This means that your applications need to support the features that help them blend into the Mac OS X ecosystem and create a seamless user experience.

Technologies

The following sections describe technologies that form a key part of the Mac OS X user experience. If you are developing an application, you should consider adopting these technologies to make sure your application integrates cleanly into Mac OS X. Most of these technologies require little effort to support but provide big advantages in your software's usability and in the likelihood of user adoption.

Aqua

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

Applications written using modern Mac OS X interfaces (such as those provided by Carbon and Cocoa) get much of the Aqua appearance automatically. However, there is more to Aqua than that. Interface designers must still follow the Aqua guidelines to position windows and controls in appropriate places. Designers must take into account features such as text, keyboard, and mouse usage and make sure their designs work appropriately for Aqua. The implementers of an interface must then write code to provide the user with appropriate feedback and to convey what is happening in an informative way.

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

Quick Look

Introduced in Mac OS X v10.5, Quick Look is a technology that enables client applications, such as Spotlight and the Finder, to display thumbnail images and full-size previews of documents. Mac OS X provides automatic support for many common content types, including HTML, RTF, plain text, TIFF, PNG, JPEG, PDF, and QuickTime movies. If your application defines custom document formats, you should provide a Quick Look generator for those formats. Generators are plug-ins that convert documents of the appropriate type from their native format to a format that Quick Look can display to users. Mac OS X makes extensive use of generators to give users quick previews of documents without having to open the corresponding applications.

For information about supporting Quick Look for your custom document types, see *Quick Look Programming Guide* and *Quick Look Framework Reference*.

Resolution-Independent User Interface

Resolution independence decouples the resolution of the user's screen from the units you use in your code's drawing operations. While Mac OS X version 10.4 and earlier assumed a screen resolution of 72 dots per inch (dpi), most modern screens actually have resolutions that are 100 dpi or more. The result of this difference is that content rendered for a 72 dpi screen appears smaller on such screens—a problem that will only get worse as screen resolutions increase.

In Mac OS X v10.4, steps were taken to support content scaling for screen-based rendering. In particular, the notion of a scale factor was introduced to the system, although not heavily publicized. This scale factor was fixed at 1.0 by default but could be changed by developers using the Quartz Debug application. In addition, Carbon and Cocoa frameworks were updated to support scale factors and interfaces were introduced to return the current screen scale factor so that developers could begin testing their applications in a content-scaled world.

Although the Mac OS X frameworks handle many aspects related to resolution-independent drawing, there are still things you need to do in your drawing code to support resolution independence:

- Update the images and artwork in your user interface. As the pixel density of displays increases, you need to make sure your application's custom artwork can scale accordingly—that is, your art needs to be larger in terms of pixel dimensions to avoid looking pixellated at higher scale factors. This includes changing:
  - Application icons
  - Images that appear in buttons or other controls
  - Other custom images you use in your interface
- Update code that relies on precise pixel alignment to take the current scale factor into account. Both Cocoa and Carbon provide ways to access the current scale factor.
- Consider drawing lines, fills, and gradients programmatically instead of using prerendered images. Shapes drawn using Quartz and Cocoa always scale appropriately to the screen resolution.

When scaling your images, be sure to cache the scaled versions of frequently-used images to increase drawing efficiency. For more information about resolution-independence and how to support it in your code, see *Resolution Independence Guidelines*.

Spotlight

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

Application developers use Spotlight in two different ways. First, you can search for file attributes and content using the Spotlight search API. Second, if your application defines its own custom file formats, you should incorporate any appropriate metadata information in those formats and provide a Spotlight importer plug-in to return that metadata to Spotlight.

Note: You should not use Spotlight for indexing and searching the general content of a file. Spotlight is intended for searching only the meta information associated with files. To search the actual contents of a file, use the Search Kit API. For more information, see "Search Kit Framework" (page 67).

In Mac OS X v10.5 and later, several new features were added to make working with Spotlight easier. The File manager includes functions for swapping the contents of a file while preserving its original metadata; see the `Files.h` header file in the Core Services framework. Spotlight also defines functions for storing lineage information with a file so that you can track modifications to that file.

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

Bundles and Packages

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

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

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

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

Code Signing

In Mac OS X v10.5 and later, it is possible to associate a digital signature with your application using the `codesign` command-line tool. If you have a certificate that is authorized for signing, you can use that certificate to sign your application's code file. Signing your application makes it possible for Mac OS X to verify the source of the application and ensure the application has not changed since it was shipped. If the application has been tampered with, Mac OS X detects the change and can alert the user to the problem. Signed applications also make it harder to circumvent parental controls and other protection features of the system.

For information on signing your application, see *Code Signing Guide*.

Internationalization and Localization

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

Internationalizing an application involves the following steps:

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

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

Software Configuration

Mac OS X programs commonly use property list files (also known as plist files) to store configuration data. A property list is a text or binary file used to manage a dictionary of key-value pairs. Applications use a special type of property list file, called an information property list (`Info.plist`) file, to communicate key attributes of the application to the system, such as the application's name, unique identification string, and version information. Applications also use property list files to store user preferences or other custom configuration data. If your application stores custom configuration data, you should consider using property lists files as well.

The advantage of property list files is that they are easy to edit and modify from outside the runtime environment of your application. Mac OS X provides several tools for creating and modifying property list files. The Property List Editor application that comes with Xcode is the main application for editing the contents of property lists. Xcode also provides a custom interface for editing your application's `Info.plist` file. (For information about information property lists files and the keys you put in them, see *Runtime Configuration Guidelines*.)

Inside your program, you can read and write property list files programmatically using facilities found in both Core Foundation and Cocoa. For more information on creating and using property lists programmatically, see *Property List Programming Guide* or *Property List Programming Topics for Core Foundation*.

Fast User Switching

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

When fast user switching is enabled, an application must be careful not to do anything that might affect another version of that application running in a different user's session. In particular, your application should avoid using or creating any shared resources unless those resources are associated with a particular user session. As you design your application, make sure that any shared resources you use are protected appropriately. For more information on how to do this, see *Multiple User Environments*.

Spaces

Introduced in Mac OS X version 10.5, Spaces lets the user organize windows into groups and switch back and forth between groups to avoid cluttering up the desktop. Most application windows appear in only one space at a time, but there may be times when you need to share a window among multiple spaces. For example, if your application has a set of shared floating palettes, you might need those palettes to show up in every space containing your application's document windows.

Cocoa provides support for sharing windows across spaces through the use of collection behavior attributes on the window. For information about setting these attributes, see *NSWindow Class Reference*.

Accessibility

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

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

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

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

For more information about accessibility, see *Accessibility Overview*.

AppleScript

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

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

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

System Applications

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

The Finder

The Finder has many functions in the operating system:

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

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

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

For information about the Finder and its relationship to the file system, see *File System Overview*.

The Dock

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

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

Dashboard

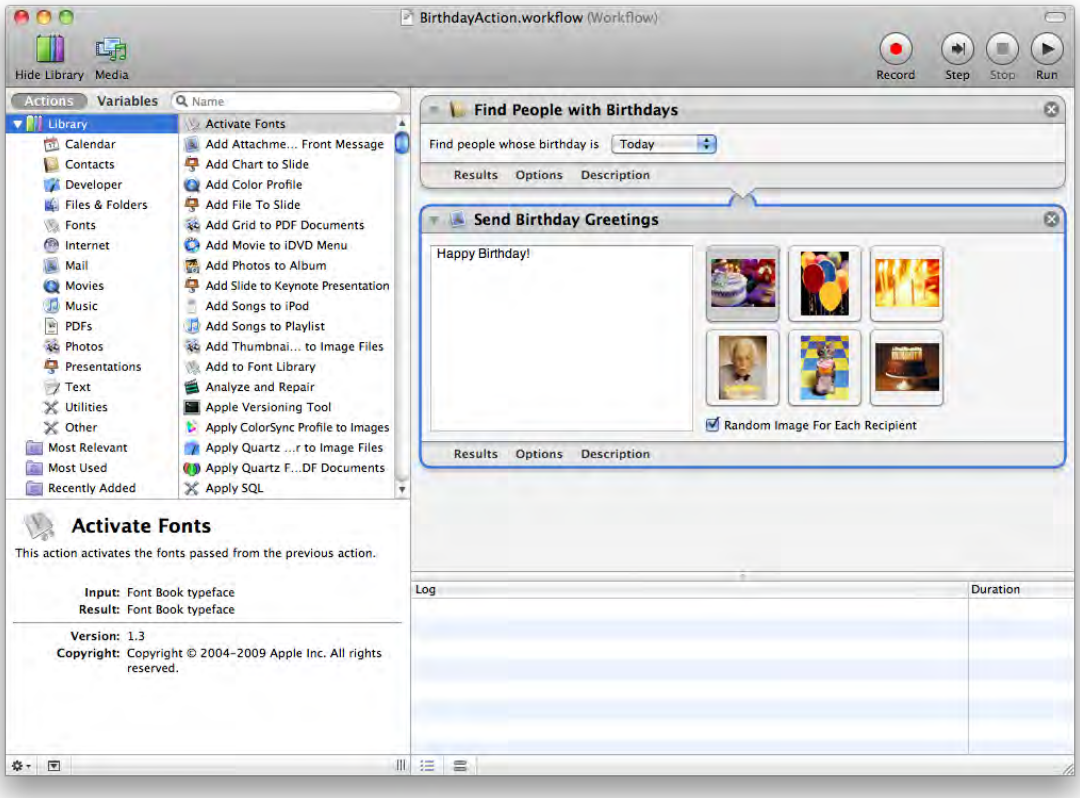
Introduced in Mac OS X v10.4, Dashboard provides a lightweight desktop environment for running widgets. Widgets are lightweight web applications that display information a user might use occasionally. You can write widgets to track stock quotes, view the current time, or access key features of a frequently used application. Widgets reside in the Dashboard layer, which is activated by the user and comes into the foreground in a manner similar to Exposé. Mac OS X comes with several standard widgets, including a calculator, clock, and iTunes controller.

For information about developing Dashboard widgets, see "Dashboard Widgets" (page 83).

Automator

Introduced in Mac OS X version 10.4, Automator lets you automate common workflows on your computer without writing any code. The workflows you create can take advantage of many features of Mac OS X and any standard applications for which predefined **actions** are available. Actions are building blocks that represent tangible tasks, such as opening a file, saving a file, applying a filter, and so on. The output from one action becomes the input to another and you assemble the actions graphically with the Automator application. Figure 5-1 shows the Automator main window and a workflow containing some actions.

Figure 5-1 Automator main window



In cases where actions are not available for the tasks you want, you can often create them yourself. Automator supports the creation of actions using Objective-C code or AppleScript. You can also create actions that are based on shell scripts, Perl, and Python.

In Mac OS X v10.5 and later, Automator supports the "Watch Me Do" feature, which lets you build an action by recording your interactions with Mac OS X and any open applications. You can use workflow variables as placeholders for dynamically changing values or pieces of text in your script. You can also integrate workflows into your applications using the classes of the Automator framework.

For more information about using Automator, see the Automator Help. For information on how to create new Automator actions, see *Automator Programming Guide*. For information about how to integrate workflows into your applications, see the classes in *Automator Framework Reference*.

Time Machine

Introduced in Mac OS X v10.5, Time Machine is an application that automatically and transparently backs up the user's files to a designated storage system. Time Machine integrates with the Finder to provide an intuitive interface for locating lost or old versions of files quickly and easily. Time Machine also provides an interface that applications can use to exclude files that should not be backed up. For more information on using this interface, see "Time Machine Support" (page 70).

Software Development Overview

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

Applications

Applications are by far the predominant type of software created for Mac OS X, or for any platform. Mac OS X provides numerous environments for developing applications, each of which is suited for specific types of development. For information about these environments and the technologies you can use to build your applications, see "Application Technologies" (page 57).

Important: You should always create universal binaries for Carbon, Cocoa, and BSD applications. Java and WebObjects may also need to create universal binaries for bridged code. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Frameworks

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

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

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

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

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

Important: You should always create universal binaries for frameworks written with Carbon, Cocoa, or BSD APIs. For information on how to create universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

Plug-ins

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

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

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

Address Book Action Plug-Ins

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

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

Application Plug-Ins

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

Automator Plug-Ins

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

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

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

Contextual Menu Plug-Ins

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

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

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

Core Audio Plug-Ins

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

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

Image Units

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

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

Input Method Components

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

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

In Mac OS X v10.5 and later, you can create input methods using the Input Method Kit (`InputMethodKit.framework`). For information on how to use this framework, see *Input Method Kit Framework Reference*. For information on how to create an input method in earlier versions of Mac OS X, see the *BasicInputMethod* sample code and the *Component Manager Reference*.

Interface Builder Plug-Ins

If you create any custom controls for your application, you can create an Interface Builder plug-in to make those controls available in the Interface Builder design environment. Creating plug-ins for your controls lets you go back and redesign your application's user interface using your actual controls, as opposed to generic custom views. For controls that are used frequently in your application, being able to see and manipulate your controls directly can eliminate the need to build your application to see how your design looks.

In Mac OS X v10.5, you should include plug-ins for any of your custom controls inside the framework bundle that implements those controls. Bundling your plug-in with your framework is not required but does make it easier for users. When the user adds your framework to their Xcode project, Interface Builder automatically scans the framework and loads the corresponding plug-in if it is present. If you did not use a framework for the implementation of your controls, you must distribute the plug-in yourself and instruct users to load it using the Interface Builder preferences window.

For information on how to create plug-ins that support Interface Builder 3.0 and later, see *Interface Builder Plug-In Programming Guide* and *Interface Builder Kit Framework Reference*. For information on how to create plug-ins for earlier versions of Interface Builder, see the header files for Interface Builder framework (`InterfaceBuilder.framework`) or the examples in <*Xcode*>`/Examples/Interface Builder`.

Metadata Importers

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

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

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

QuickTime Components

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

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

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

Safari Plug-ins

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

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

Dashboard Widgets

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

Creating widgets is simpler than creating most applications because widgets are effectively HTML-based applications with optional JavaScript code to provide dynamic behavior. Dashboard uses WebKit to provide the environment for displaying the HTML and running the JavaScript code. Your widgets can take advantage

of several extensions provided by that environment, including a way to render content using Quartz-like JavaScript functions. In Mac OS X v10.5 and later, you can create widgets using the Dashcode application, which is described in "Dashcode" (page 134).

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

Agent Applications

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

An agent may be launched by the user but is more likely to be launched by the system or another application. As a result, agents do not show up in the Dock or the Force Quit window. Agents also do not have a menu bar for choosing commands. User manipulation of an agent typically occurs through dialogs or contextual menus in the agent user interface. For example, the iChat application uses an agent to communicate with the chat server and notify the user of incoming chat requests. The Dock is another agent program that is launched by the system for the benefit of the user.

The way to create an agent application is to create a bundled application and include the `LSUIElement` key in its `Info.plist` file. The `LSUIElement` key notifies the Dock that it should treat the application as an agent when double-clicked by the user. For more information on using this key, see *Runtime Configuration Guidelines*.

Screen Savers

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

Slideshows

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

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

Programmatic Screen Savers

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

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

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

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

Services

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

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

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

Preference Panes

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

You might need to create preference panes if you create:

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

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

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

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

Web Content

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

In addition to "WebObjects" (page 59) and "Dashboard Widgets" (page 83), the following sections list ways to deliver web content in Mac OS X. For more information about developing web content, see *Getting Started with Internet and Web*.

Dynamic Websites

Mac OS X provides support for creating and testing dynamic content in web pages. If you are developing CGI-based web applications, you can create websites using a variety of scripting technologies, including Perl and PHP. A complete list of scripting technologies is provided in "Scripts" (page 88). You can also create and deploy more complex web applications using JBoss, Tomcat, and WebObjects. To deploy your webpages, use the built-in Apache web server.

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

SOAP and XML-RPC

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

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

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

Sherlock Channels

In Mac OS X v10.4 and earlier, the Sherlock application was a host for Sherlock channels. A Sherlock channel is a developer-created module that combines web services with an Aqua interface to provide a unique way for users to find information. Sherlock channels combined related, but different, types of information in one window.

Sherlock channels are not supported in Mac OS X v10.5 and later.

Mail Stationery

The Mail application in Mac OS X v10.5 and later supports the creation of email messages using templates. Templates provide the user with prebuilt email messages that can be customized quickly before being sent. Because templates are HTML-based, they can incorporate images and advanced formatting to give the user's email a much more stylish and sophisticated appearance.

Developers and web designers can create custom template packages for external or internal users. Each template consists of an HTML page, property list file, and images packaged together in a bundle, which is then stored in the Mail application's stationery directory. The HTML page and images define the content of the email message and can include drop zones for custom user content. The property list file provides Mail with information about the template, such as its name, ID, and the name of its thumbnail image.

For information about how to create new stationery templates, see *Mail Programming Topics*.

Command-Line Tools

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

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

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

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

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

Launch Items, Startup Items, and Daemons

Launch items and startup items are special programs that launch other programs or perform one-time operations during startup and login periods. Daemons are programs that run continuously and act as servers for processing client requests. You typically use launch items and startup items to launch daemons or perform periodic maintenance tasks, such as checking the hard drive for corrupted information. Launch items run under the `launchd` system process and are supported only in Mac OS X v10.4 and later. Startup items are also used to launch system and user-level processes but are deprecated in current versions of Mac OS X. They may be used to launch daemons and run scripts in Mac OS X v10.3.9 and earlier.

Launch items and startup items should not be confused with the login items found in the Accounts system preferences. Login items are typically agent applications that run within a given user's session and can be configured by that user. Launch items and startup items are not user-configurable.

Few developers should ever need to create launch items or daemons. They are reserved for the special case where you need to guarantee the availability of a particular service. For example, Mac OS X provides a launch item to run the DNS daemon. Similarly, a virus-detection program might install a launch item to launch a daemon that monitors the system for virus-like activity. In both cases, the launch item would run its daemon in the root session, which provides services to all users of the system.

For more information about launch items, startup items, and daemons, see *System Startup Programming Topics*.

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

Scripts

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

Table 6-1 lists many of the scripting languages supported by Mac OS X along with a description of the strengths of each language.

Table 6-1 Scripting language summary

Script language	Description
AppleScript	An English-based language for controlling scriptable applications in Mac OS X. Use it to tie together applications involved in a custom workflow or repetitive job. See *AppleScript Overview* for more information.
`bash`	A Bourne-compatible shell script language used to build programs on UNIX-based systems.
`csh`	The C shell script language used to build programs on UNIX-based systems.
Perl	A general-purpose scripting language supported on many platforms. It comes with an extensive set of features suited for text parsing and pattern matching and also has some object-oriented features. See http://www.perl.org/ for more information.
PHP	A cross-platform, general-purpose scripting language that is especially suited for web development. See http://www.php.net/ for more information.
Python	A general-purpose, object-oriented scripting language implemented for many platforms. See http://www.python.org/ for more information. In Mac OS X v10.4 and later, you can also use Python with the Cocoa scripting bridge; see *Ruby and Python Programming Topics for Mac OS X*.
Ruby	A general-purpose, object-oriented scripting language implemented for many platforms. See http://www.ruby-lang.org/ for more information. In Mac OS X v10.5 and later, you can also use Ruby with the Cocoa scripting bridge; see *Ruby and Python Programming Topics for Mac OS X*.
`sh`	The Bourne shell script language used to build programs on UNIX-based systems.
Tcl	Tool Command Language. A general-purpose language implemented for many platforms. It is often used to create graphical interfaces for scripts. See http://www.tcl.tk/ for more information.
`tcsh`	A variant of the C shell script language used to build programs on UNIX-based systems.
`zsh`	The Z shell script language used to build programs on UNIX-based systems.

For introductory material on using the command line, see "Command Line Primer" (page 109).

Scripting Additions for AppleScript

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

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

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

Kernel Extensions

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

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

Kernel extensions are typically used to implement new network stacks or file systems. You would not use kernel extensions to communicate with external devices such as digital cameras or printers. (For information on communicating with external devices, see "Device Drivers" (page 90).)

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

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

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

Device Drivers

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

Although developers of mice and keyboards may be able to use the standard drivers, many other developers require custom drivers. Developers of hardware such as scanners, printers, AGP cards, and PCI cards typically have to create custom drivers for their devices. These devices require more sophisticated data handling than

is usually needed for mice and keyboards. Hardware developers also tend to differentiate their hardware by adding custom features and behavior, which makes it difficult for Apple to provide generic drivers to handle all devices.

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

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

Choosing Technologies to Match Your Design Goals

Mac OS X has many layers of technology. Before choosing a specific technology to implement a solution, think about the intended role for that technology. Is that technology appropriate for your needs? Is there a better technology available? In some cases, Mac OS X offers several technologies that implement the same behavior but with varying levels of complexity and flexibility. Understanding your operational needs can help you make appropriate choices during design.

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

High Performance

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

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

Table 7-1 lists several Mac OS X technologies that you can use to improve the performance of your software.

Table 7-1 Technologies for improving performance

Technology	Description
`NSOperation` and `NSOperationQueue`	Mac OS X v10.5 includes two new Cocoa classes that simplify the process of supporting multiple threads in your application. The `NSOperation` object acts as a wrapper for encapsulated tasks while the `NSOperationQueue` object manages the execution of those tasks. Operations support dependency and priority ordering and can be customized to configure the threading environment as needed. For more information about these classes, see *Concurrency Programming Guide*.
Grand Central Dispatch	Introduced in Mac OS X v10.6, Grand Central Dispatch provides a new model for increasing the amount of concurrent work performed by your application. In this model, you factor your application's tasks into discrete chunks and submit them to a dispatch queue. The dispatch queue then works with the system to execute as many tasks as possible in the most efficient way possible. For more information about using dispatch queues, see *Concurrency Programming Guide*.

Technology	Description
OpenCL	Introduced in Mac OS X v10.6, OpenCL provides a way for you to distribute tasks among the available GPUs and CPUs on the target system. Given the amount of processing power available in modern GPUs, this feature can improve the performance of data-parallel tasks significantly. For information on how to use OpenCL, see *OpenCL Programming Guide for Mac OS X*.
64-bit	Although not appropriate in all cases, providing a 64-bit version of your application can improve performance, especially on Intel-based Macintosh computers. The 64-bit capable Intel processors typically have more hardware registers available for performing calculations and passing function parameters. More registers often leads to better performance. As always, test your code in both 32-bit and 64-bit modes to see if providing a 64-bit version is worthwhile. For more information, see *64-Bit Transition Guide*.
Accelerate Framework	The Accelerate framework provides an API for performing multiple scalar or floating-point operations in parallel by taking advantage of the underlying processor's vector unit. Because it is tuned for both PowerPC and Intel processor architectures, using the Accelerate framework eliminates the need for you to write custom code for both the AltiVec and SSE vector units. For more information about using this framework, see *Accelerate Release Notes*.
Instruments and Shark	Apple provides a suite of performance tools for measuring many aspects of your software. Instruments and Shark in particular provide new ways of looking at your application while it runs and analyzing its performance. Use these tools to identify hot spots and gather performance metrics that can help identify potential problems. For more information Instruments, see "Instruments" (page 135). For information about Shark and the other performance tools that come with Mac OS X, see "Performance Tools" (page 148).
Lower-level APIs	Mac OS X provides many layers of APIs. As you consider the design of your application, examine the available APIs to find the appropriate tradeoff between performance, simplicity, and flexibility that you need. Usually, lower-level system APIs offer the best performance but are more complicated to use. Conversely, higher-level APIs may be simpler to use but be less flexible. Whenever possible, choose the lowest-level API that you feel comfortable using.
Threads	If operations or Grand Central Dispatch do not provide the support you need, you can still use threads to increase the amount of concurrent work performed by your application. Mac OS X implements user-level threads using the POSIX threading package but also supports several higher-level APIs for managing threads. For information about these APIs and threading support in general, see "Threading Support" (page 29) and *Threading Programming Guide*.

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

Note: For specific information about legacy Carbon managers and the recommended replacements for them, see "Carbon Considerations" (page 105).

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

Easy to Use

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

Table 7-2 lists several Mac OS X technologies that you can use to make your software easier to use.

Table 7-2 Technologies for achieving ease of use

Technology	Description
Aqua	If your program has a visual interface, it should adhere to the human interface guidelines for Aqua, which include tips for how to lay out your interface and manage its complexity. For more information, see "Aqua" (page 71).
Quick Look	Introduced in Mac OS X v10.5, Quick Look generates previews of user documents that can be displayed in the Finder and Dock. These previews make it easier for the user to find relevant information quickly without launching any applications. For more information, see "Quick Look" (page 71).
Bonjour	Bonjour simplifies the process of configuring and detecting network services. Your program can vend network services or use Bonjour to be a client of an existing network service. For more information, see "Bonjour" (page 61).
Accessibility technologies	The Accessibility interfaces for Carbon and Cocoa make it easier for people with disabilities to use your software. For more information, see "Accessibility" (page 75).
AppleScript	AppleScript makes it possible for users to automate complex workflows quickly. It also gives users a powerful tool for controlling your application. For more information, see "AppleScript" (page 76).
Internationalization	Mac OS X provides significant infrastructure for internationalizing software bundles. For more information, see "Internationalization and Localization" (page 74).

Technology	Description
Keychain Services	Keychains provide users with secure access to passwords, certificates, and other secret information. Adding support for Keychain Services in your program can reduce the number of times you need to prompt the user for passwords and other secure information. For more information, see "Keychain Services" (page 65).

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

Attractive Appearance

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

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

Table 7-3 lists several Mac OS X technologies you can use to ensure that your software has an attractive appearance.

Table 7-3 Technologies for achieving an attractive appearance

Technology	Description
Aqua	Aqua defines the guidelines all developers should follow when crafting their application's user interface. Following these guidelines ensures that your application looks and feels like a Mac OS X application. For more information, see "Aqua" (page 71).
Resolution independence	Screen resolutions continue to increase with most screens now supporting over 100 pixels per inch. In order to prevent content from shrinking too much, Mac OS X will soon apply a scaling factor to drawing operations to keep them at an appropriate size. Your software needs to be ready for this scaling factor by being able to draw more detailed content in the same "logical" drawing area. For more information, see "Resolution-Independent User Interface" (page 72).
Core Animation	In Mac OS X v10.5 and later, you can use Core Animation to add advanced graphics behaviors to your software. Core Animation a lightweight mechanism for performing advanced animations in your Cocoa views. For more information, see "Core Animation" (page 46).
Quartz	Quartz is the native (and preferred) 2D rendering API for Mac OS X. It provides primitives for rendering text, images, and vector shapes and includes integrated color management and transparency support. For more information, see "Quartz" (page 43).

Technology	Description
Core Text	In Mac OS X v10.5 and later, Core Text replaces the ATSUI and MLTE technologies as the way to high quality rendering and layout of Unicode text for Carbon and Cocoa applications. The Cocoa text system uses Core Text for its implementation. For more information, see "Core Text" (page 49).
Core Image	In Mac OS X v10.4 and later, Core Image provides advanced image processing effects for your application. Core Image makes it possible to manipulate image data in real time using the available hardware rendering and to perform complex manipulations that make your application look stunning. For more information, see "Core Image" (page 47).
OpenGL	OpenGL is the preferred 3D rendering API for Mac OS X. The Mac OS X implementation of OpenGL is hardware accelerated on many systems and has all of the standard OpenGL support for shading and textures. See *OpenGL Programming Guide for Mac OS X* for an overview of OpenGL and guidelines on how to use it. For an example of how to use OpenGL with Cocoa, see the sample code project *Cocoa OpenGL*.

Reliability

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

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

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

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

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

All of the technologies of Mac OS X offer a high degree of reliability. However, Table 7-4 lists some specific technologies that improve reliability by reducing the amount of complex code you have to write from scratch.

Table 7-4 Technologies for achieving reliability

Technology	Description
Code signing	Code signing associates a digital signature with your application and helps the system determine when your application has changed, possibly because of tampering. When changes occur, the system can warn the user and provide an option for disabling the application. For more information, see "Code Signing" (page 73).
Authorization Services	Authorization Services provides a way to ensure that only authorized operations take place. Preventing unauthorized access helps protect your program as well as the rest of the system. See *Authorization Services Programming Guide* for more information.
Core Foundation	Core Foundation supports basic data types and eliminates the need for you to implement string and collection data types, among others. Both Carbon and Cocoa support the Core Foundation data types, which makes it easier for you to integrate them into your own data structures. See *Getting Started with Core Foundation* for more information.
WebKit	WebKit provides a reliable, standards-based mechanism for rendering HTML content (including JavaScript code) in your application.

Adaptability

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

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

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

Table 7-5 Technologies for achieving adaptability

Technology	Description
FSEvents API	The FSEvents API lets you detect changes to the file system easily and efficiently. You might use this technology to update your application's internal data structures whenever changes occur to specific directories or directory hierarchies. For more information, see "FSEvents API" (page 33).
Core Foundation	Core Foundation provides services for managing date, time, and number formats based on any locale. See Reference Library > Core Foundation for specific reference documents.

Technology	Description
Quartz Services	Quartz Services provides access to screen information and provides notifications when screen information changes. See *Quartz Display Services Reference* for more information.
Bonjour	Bonjour simplifies the process of configuring and detecting network services. Your program can vend network services or use Bonjour to be a client of an existing network service. For more information, see "Bonjour" (page 61).
System Configuration	The System Configuration framework provides information about availability of network entities. See *System Configuration Framework Reference* and *System Configuration Programming Guidelines* for more information.

Interoperability

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

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

Table 7-6 lists some Mac OS X technologies that you can use to improve the interoperability of your software.

Table 7-6 Technologies for achieving interoperability

Technology	Description
AppleScript	AppleScript is a scripting system that gives users direct control over your application as well as parts of Mac OS X. See *AppleScript Overview* for information on supporting AppleScript.
Drag and drop	Although primarily implemented in applications, you can add drag and drop support to any program with a user interface. See *Drag Manager Reference* or *Drag and Drop Programming Topics for Cocoa* for information on how to integrate drag and drop support into your program.
Pasteboard	Both Carbon and Cocoa support cut, copy, and paste operations through the pasteboard. See the `Pasteboard.h` header file in the HIServices framework or *Pasteboard Programming Topics for Cocoa* for information on how to support the pasteboard in your program.
Bonjour	Your program can vend network services or use Bonjour to be a client of an existing network service. For more information, see "Bonjour" (page 61).

Technology	Description
Services	Services let the user perform a specific operation in your application using data on the pasteboard. Services use the pasteboard to exchange data but act on that data in a more focused manner than a standard copy-and-paste operation. For example, a service might create a new mail message and paste the data into the message body. See *Setting Up Your Carbon Application to Use the Services Menu* or *Services Implementation Guide* for information on setting up an application to use services.
XML	XML is a structured format that can be used for data interchange. Mac OS X provides extensive support for reading, writing, and parsing XML data. For more information, see "XML Parsing Libraries" (page 70).

Mobility

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

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

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

Table 7-7 Technologies for achieving mobility

Technology	Description
Performance	An efficient application uses fewer instructions to compute its data. On portable computers, this improved efficiency translates to power savings and a longer battery life. You should strive to make your applications as efficient as possible using the available system technologies and tools. For more information, see "High Performance" (page 93).
CFNetwork	CFNetwork provides a modern interface for accessing network services and handling changes in the network configuration. See *CFNetwork Programming Guide* for an introduction to the CFNetwork API.
Quartz Services	Quartz Services provides access to screen information and provides notifications when screen information changes. See *Quartz Display Services Reference* for information about the API.
Bonjour	Bonjour lets mobile users find services easily or vend their own services for others to use. For more information, see "Bonjour" (page 61).

Technology	Description
System Configuration	The System Configuration framework is the foundation for Apple's mobility architecture. You can use its interfaces to get configuration and status information for network entities. It also sends out notifications when the configuration or status changes. See *System Configuration Programming Guidelines* for more information.

Porting Tips

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

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

64-Bit Considerations

With Macintosh computers using 64-bit PowerPC and Intel processors, developers can begin writing software to take advantage of the 64-bit architecture provided by these chips. For many developers, however, compiling their code into 64-bit programs may not offer any inherent advantages. Unless your program needs more than 4 GB of addressable memory, supporting 64-bit pointers may only reduce the performance of your application.

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

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

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

There are several documents to help you create 64-bit applications. For general information about making the transition, see *64-Bit Transition Guide*. For Cocoa-specific information, see *64-Bit Transition Guide for Cocoa*. For Carbon-specific information, see *64-Bit Guide for Carbon Developers*.

.

Windows Considerations

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

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

- **Avoid custom controls**. Avoid creating custom controls if Mac OS X already provides equivalent controls for your needs. Custom controls are appropriate only in situations where the control is unique to your needs and not provided by the system. Replacing standard controls can make your interface look out of place and might confuse users.
- **Use a single menu bar**. The Mac OS X menu bar is always at the top of the screen and always contains the commands for the currently active application. You should also pay attention to the layout and placement of menu bar commands, especially commonly used commands such as New, Open, Quit, Copy, Minimize, and Help.
- **Pay attention to keyboard shortcuts**. Mac OS X users are accustomed to specific keyboard shortcuts and use them frequently. Do not simply migrate the shortcuts from your Windows application to your Mac OS X application. Also remember that Mac OS X uses the Command key not the Control key as the main keyboard modifier.
- **Do not use MDI**. The Multiple Document Interface (MDI) convention used in Microsoft Windows directly contradicts Mac OS X design guidelines. Windows in Mac OS X are document-centric and not application-centric. Furthermore, the size of a document window is constrained only by the user's desktop size.
- **Use Aqua**. Aqua gives Mac OS X applications the distinctive appearance and behavior that users expect from the platform. Using nonstandard layouts, conventions, or user interface elements can make your application seem unpolished and unprofessional.
- **Design high-quality icons and images**. Mac OS X icons are often displayed in sizes varying from 16x16 to 512x512 pixels. These icons are usually created professionally, with millions of colors and photo-realistic qualities. Your application icons should be vibrant and inviting and should immediately convey your application's purpose.
- **Design clear and consistent dialogs**. Use the standard Open, Save, printing, Colors, and Font dialogs in your applications. Make sure alert dialogs follow a consistent format, indicating what happened, why it happened, and what to do about it.
- **Consider toolbars carefully**. Having a large number of buttons, especially in an unmovable toolbar, contributes to visual clutter and should be avoided. When designing toolbars, include icons only for menu commands that are not easily discoverable or that may require multiple clicks to be reached.
- **Use an appropriate layout for your windows**. The Windows user interface relies on a left-biased, more crowded layout, whereas Aqua relies on a center-biased, spacious layout. Follow the Aqua guidelines to create an appealing and uncluttered interface that focuses on the task at hand.

- **Avoid application setup steps**. Whenever possible, Mac OS X applications should be delivered as drag-and-drop packages. If you need to install files in multiple locations, use an installation package to provide a consistent installation experience for the user. If your application requires complex setup procedures in order to run, use a standard Mac OS X assistant. For more information, see "Bundles and Packages" (page 73).
- **Use filename extensions**. Mac OS X fully supports and uses filename extensions. For more information about filename extensions, see *File System Overview*.

Carbon Considerations

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

Migrating From Mac OS 9

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

Required Replacement Technologies

The technologies listed in Table 8-1 cannot be used in Carbon. You must use the technology in the "Now use" column instead.

Table 8-1 Required replacements for Carbon

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

Recommended Replacement Technologies

The technologies listed in Table 8-2 can still be used in Carbon, but the indicated replacements provide more robust support and are preferred.

Table 8-2 Recommended replacements for Carbon

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

Use the Carbon Event Manager

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

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

Use the HIToolbox

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

Note: The HIToolbox interfaces are available for creating 32-bit applications only. If you are creating 64-bit applications, you should use Cocoa for your user interface instead.

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

Use Nib Files

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

For information about using Interface Builder, see *Interface Builder User Guide*.

Command Line Primer

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

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

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

Basic Shell Concepts

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

Getting Information

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

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

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

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

Shell	Command
`bash`	`help` or `bash`

Shell	Command
sh	help or sh
csh	builtins or csh
tcsh	builtins or tcsh
zsh	zshbuiltins

Specifying Files and Directories

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

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

Table A-2 Special path characters and their meaning

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

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

Accessing Files on Volumes

On a typical UNIX system, the storage provided by local disk drives is coalesced into a single monolithic file system with a single root directory. This differs from the way the Finder presents local disk drives, which is as one or more volumes, with each volume acting as the root of its own directory hierarchy. To satisfy both worlds, Mac OS X includes a hidden directory Volumes at the root of the local file system. This directory contains all of the volumes attached to the local computer. To access the contents of other local volumes,

you should always add the volume path at the beginning of the remaining directory information. For example, to access the `Applications` directory on a volume named `MacOSX`, you would use the path `/Volumes/MacOSX/Applications`

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

Flow Control

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

Table A-3 Input and output sources for programs

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

Redirecting Input and Output

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

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

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

Terminating Programs

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

Frequently Used Commands

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

Table A-4 lists some of the more commonly used commands and programs. Because most of the items in this table are not built-in shell commands, you can use them from any shell. For syntax and usage information for each command, see the corresponding man page. For a more in-depth list of commands and their accompanying documentation, see *Mac OS X Man Pages*.

Table A-4 Frequently used commands and programs

Command	Meaning	Description
`cat`	Catenate	Catenates the specified list of files to `stdout`.
`cd`	Change Directory	A common shell command used to navigate the directory hierarchy.
`cp`	Copy	Copies files and directories (using the -r option) from one location to another.
`date`	Date	Displays the current date and time using the standard format. You can display this information in other formats by invoking the command with specific arguments.
`echo`	Echo to Output	Writes its arguments to `stdout`. This command is most often used in shell scripts to print status information to the user.
`less`	Scroll Through Text	Used to scroll through the contents of a file or the results of another shell command. This command allows forward and backward navigation through the text.
`ls`	List	Displays the contents of the current directory. Specify the `-a` argument to list all directory contents (including hidden files and directories). Use the `-l` argument to display detailed information for each entry.
`mkdir`	Make Directory	Creates a new directory.
`more`	Scroll Through Text	Similar to the `less` command but more restrictive. Allows forward scrolling through the contents of a file or the results of another shell command.
`mv`	Move	Moves files and directories from one place to another. You also use this command to rename files and directories.

Command	Meaning	Description
`open`	Open an application or file.	You can use this command to launch applications from Terminal and optionally open files in that application.
`pwd`	Print Working Directory	Displays the full path of the current directory.
`rm`	Remove	Deletes the specified file or files. You can use pattern matching characters (such as the asterisk) to match more than one file. You can also remove directories with this command, although use of `rmdir` is preferred.
`rmdir`	Remove Directory	Deletes a directory. The directory must be empty before you delete it.
Ctrl-C	Abort	Sends an abort signal to the current command. In most cases this causes the command to terminate, although commands may install signal handlers to trap this command and respond differently.

Environment Variables

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

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

```
% export MYFUNCTION=MyGetData
```

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

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

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

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

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

Running Programs

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

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

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

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

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

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

Mac OS X Frameworks

This appendix contains information about the frameworks of Mac OS X. These frameworks provide the interfaces you need to write software for the platform. Some of these frameworks contain simple sets of interfaces while others contain multiple subframeworks. Where applicable, the tables in this appendix list any key prefixes used by the classes, methods, functions, types, or constants of the framework. You should avoid using any of the specified prefixes in your own symbol names.

System Frameworks

Table B-1 describes the frameworks located in the `/System/Library/Frameworks` directory and lists the first version of Mac OS X in which each became available.

Table B-1 System frameworks

Name	First available	Prefixes	Description
`Accelerate.framework`	10.3	`cblas`, `vDSP`, `vv`	Umbrella framework for vector-optimized operations. See "Accelerate Framework" (page 121).
`AddressBook.framework`	10.2	`AB`, `ABV`	Contains functions for creating and accessing a systemwide database of contact information.
`AGL.framework`	10.0	`AGL`, `GL`, `glm`, `GLM`, `glu`, `GLU`	Contains Carbon interfaces for OpenGL.
`AppKit.framework`	10.0	`NS`	Contains classes and methods for the Cocoa user-interface layer. In general, link to `Cocoa.framework` instead of this framework.
`AppKit-Scripting.framework`	10.0	N/A	Deprecated. Use `AppKit.framework` instead.
`AppleScriptKit.framework`	10.0	`ASK`	Contains interfaces for creating AppleScript plug-ins.
`AppleScriptObj-C.framework`	10.6	`NS`	Contains Objective-C extensions for creating AppleScript plug-ins.
`AppleShare-Client.framework`	10.0	`AFP`	Deprecated. Use `NetFS.framework` instead.

Name	First available	Prefixes	Description
AppleShareClient-Core.framework	10.0	AFP	Contains utilities for handling URLs in AppleShare clients.
AppleTalk.framework	10.0	N/A	Deprecated. Do not use.
Application-Services.framework	10.0	AE, AX, ATSU, CG, CT, LS, PM, QD, UT	Umbrella framework for several application-level services. See "Application Services Framework" (page 121).
AudioToolbox.framework	10.0	AU, AUMIDI	Contains interfaces for getting audio stream data, routing audio signals through audio units, converting between audio formats, and playing back music.
AudioUnit.framework	10.0	AU	Contains interfaces for defining Core Audio plug-ins.
Automator.framework	10.4	AM	Umbrella framework for creating Automator plug-ins. See "Automator Framework" (page 122).
CalendarStore.framework	10.5	Cal	Contains interfaces for managing iCal calendar data.
Carbon.framework	10.0	HI, HR, ICA, ICD, Ink, Nav, OSA, PM, SFS,SR	Umbrella framework for Carbon-level services. See "Carbon Framework" (page 122).
Cocoa.framework	10.0	NS	Wrapper for including the Cocoa frameworks AppKit.framework, Foundation.framework, and CoreData.framework.
Collaboration.framework	10.5	CB	Contains interfaces for managing identity information.
CoreAudio.framework	10.0	Audio	Contains the hardware abstraction layer interface for manipulating audio.
CoreAudioKit.framework	10.4	AU	Contains Objective-C interfaces for audio unit custom views.
CoreData.framework	10.4	NS	Contains interfaces for managing your application's data model.
CoreFoundation.framework	10.0	CF	Provides fundamental software services, including abstractions for common data types, string utilities, collection utilities, plug-in support, resource management, preferences, and XML parsing.

Name	First available	Prefixes	Description
CoreLocation.framework	10.6	CL	Provides interfaces for determining the geographical location of a computer.
CoreMIDI.framework	10.0	MIDI	Contains utilities for implementing MIDI client programs.
CoreMIDIServer.framework	10.0	MIDI	Contains interfaces for creating MIDI drivers to be used by the system.
CoreServices.framework	10.0	CF, DCS, MD, SK, WS	Umbrella framework for system-level services. See "Core Services Framework" (page 123).
CoreVideo.framework	10.5	CV	Contains interfaces for managing video-based content.
CoreWLAN.framework	10.6	CW	Contains interfaces for managing wireless networks.
DirectoryService.framework	10.0	ds	Contains interfaces for supporting network-based lookup and directory services in your application. You can also use this framework to develop directory service plug-ins.
DiscRecording.framework	10.2	DR	Contains interfaces for burning data to CDs and DVDs.
DiscRecordingUI.framework	10.2	DR	Contains the user interface layer for interacting with users during the burning of CDs and DVDs.
DiskArbitration.framework	10.4	DA	Contains interfaces for monitoring and responding to hard disk events.
DrawSprocket.framework	10.0	DSp	Contains the game sprocket component for drawing content to the screen.
DVComponentGlue.framework	10.0	IDH	Contains interfaces for communicating with digital video devices, such as video cameras.
DVDPlayback.framework	10.3	DVD	Contains interfaces for embedding DVD playback features into your application.
ExceptionHandling.framework	10.0	NS	Contains exception-handling classes for Cocoa applications.
ForceFeedback.framework	10.2	FF	Contains interfaces for communicating with force feedback–enabled devices.
Foundation.framework	10.0	NS	Contains the classes and methods for the Cocoa Foundation layer. If you are creating a Cocoa application, linking to the Cocoa framework is preferable.

Name	First available	Prefixes	Description
FWAUserLib.framework	10.2	FWA	Contains interfaces for communicating with FireWire-based audio devices.
GLUT.framework	10.0	glut, GLUT	Contains interfaces for the OpenGL Utility Toolkit, which provides a platform-independent interface for managing windows.
ICADevices.framework	10.3	ICD	Contains low-level interfaces for communicating with digital devices such as scanners and cameras. See also, "Carbon Framework" (page 122).
ImageCapture-Core.framework	10.6	IC	Contains Objective-C interfaces for communicating with digital devices such as scanners and cameras.
IMCore.framework	10.6	IM	Do not use.
InputMethodKit.framework	10.5	IMK	Contains interfaces for developing new input methods, which are modules that handle text entry for complex languages.
Installer-Plugins.framework	10.4	IFX	Contains interfaces for creating plug-ins that run during software installation sessions.
InstantMessage.framework	10.4	FZ, IM	Contains interfaces for obtaining the online status of an instant messaging user.
IOBluetooth.framework	10.2	IO	Contains interfaces for communicating with Bluetooth devices.
IOBluetoothUI.framework	10.2	IO	Contains the user interface layer for interacting with users manipulating Bluetooth devices.
IOKit.framework	10.0	IO, IOBSD, IOCF	Contains the main interfaces for creating user-space device drivers and for interacting with kernel-resident drivers from user space.
IOSurface.framework	10.6	IO	Contains low-level interfaces for sharing graphics surfaces between applications.
JavaEmbedding.framework	10.0	N/A	Do not use.
JavaFrame-Embedding.framework	10.5	N/A	Contains interfaces for embedding Java frames in Objective-C code.
JavaScriptCore.framework	10.5	JS	Contains the library and resources for executing JavaScript code within an HTML page. (Prior to Mac OS X v10.5, this framework was part of WebKit.framework.

Name	First available	Prefixes	Description
JavaVM.framework	10.0	JAWT, JDWP, JMM, JNI, JVMDI, JVMPI, JVMTI	Contains the system's Java Development Kit resources.
Kerberos.framework	10.0	GSS, KL, KRB, KRB5	Contains interfaces for using the Kerberos network authentication protocol.
Kernel.framework	10.0	*numerous*	Contains the interfaces for kext development, including Mach, BSD, libkern, I/O Kit, and the various families built on top of I/O Kit.
LatentSemantic-Mapping.framework	10.5	LSM	Contains interfaces for classifying text based on latent semantic information.
LDAP.framework	10.0	N/A	Do not use.
Message.framework	10.0	AS, MF, PO, POP, RSS, TOC, UR, URL	Contains Cocoa extensions for mail delivery.
NetFS.framework	10.6	NetFS	Contains interfaces for working with network file systems.
OpenAL.framework	10.4	AL	Contains the interfaces for OpenAL, a cross-platform 3D audio delivery library.
OpenCL.framework	10.6	CL, cl	Contains the interfaces for distributing general-purpose computational tasks across the available GPUs and CPUs of a computer.
OpenDirectory.framework	10.6	OD	Contains Objective-C interfaces for managing Open Directory information.
OpenGL.framework	10.0	CGL, GL, glu, GLU	Contains the interfaces for OpenGL, which is a cross-platform 2D and 3D graphics rendering library.
OSAKit.framework	10.4	OSA	Contains Objective-C interfaces for managing and executing OSA-compliant scripts from your Cocoa applications.
PCSC.framework	10.0	MSC, Scard, SCARD	Contains interfaces for interacting with smart card devices.
Preference-Panes.framework	10.0	NS	Contains interfaces for implementing custom modules for the System Preferences application.
PubSub.framework	10.5	PS	Contains interfaces for subscribing to RSS and Atom feeds.

Name	First available	Prefixes	Description
Python.framework	10.3	Py	Contains the open source Python scripting language interfaces.
QTKit.framework	10.4	QT	Contains Objective-C interfaces for manipulating QuickTime content.
Quartz.framework	10.4	GF, PDF, QC, QCP	Umbrella framework for Quartz services. See "Quartz Framework" (page 124)
QuartzCore.framework	10.4	CA,CI, CV	Contains the interfaces for Core Image, Core Animation, and Core Video.
QuickLook.framework	10.5	QL	Contains interfaces for generating thumbnail previews of documents.
QuickTime.framework	10.0	N/A	Contains interfaces for embedding QuickTime multimedia into your application.
Ruby.framework	10.5	N/A	Contains interfaces for the Ruby scripting language.
RubyCocoa.framework	10.5	RB	Contains interfaces for running Ruby scripts from Objective-C code.
ScreenSaver.framework	10.0	N/A	Contains interfaces for writing screen savers.
Scripting.framework	10.0	NS	Deprecated. Use Foundation.framework instead.
Scripting-Bridge.framework	10.5	SB	Contains interfaces for running scripts from Objective-C code.
Security.framework	10.0	CSSM, Sec	Contains interfaces for system-level user authentication and authorization.
Security-Foundation.framework	10.3	Sec	Contains Cocoa interfaces for authorizing users.
Security-Interface.framework	10.3	PSA, SF	Contains the user interface layer for authorizing users in Cocoa applications.
Server-Notification.framework	10.6	NS	Contains Objective-C interfaces for sending and receiving server-based notifications.
Service-Management.framework	10.6	SM	Contains interfaces for managing the work performed by services.
SyncServices.framework	10.4	ISync	Contains the interfaces for synchronizing application data with a central database.
System.framework	10.0	N/A	Do not use.

Name	First available	Prefixes	Description
System-Configuration.framework	10.0	SC	Contains interfaces for accessing system-level configuration information.
Tcl.framework	10.3	Tcl	Contains interfaces for accessing the system's Tcl interpreter from an application.
Tk.framework	10.4	Tk	Contains interfaces for accessing the system's Tk toolbox from an application.
TWAIN.framework	10.2	TW	Contains interfaces for accessing TWAIN-compliant image-scanning hardware.
vecLib.framework	10.0	N/A	Deprecated. Use Accelerate.framework instead. See "Accelerate Framework" (page 121).
WebKit.framework	10.2	DOM, Web	Umbrella framework for rendering HTML content. See "WebKit Framework" (page 125).
Xgrid-Foundation.framework	10.4	XG	Contains interfaces for connecting to and managing computing cluster software.

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

Accelerate Framework

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

Table B-2 Subframeworks of the Accelerate framework

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

Application Services Framework

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

Table B-3 Subframeworks of the Application Services framework

Subframework	Description
`ATS.framework`	Contains interfaces for font layout and management using Apple Type Services.
`ColorSync.framework`	Contains interfaces for color matching using ColorSync.
`CoreGraphics.framework`	Contains the Quartz interfaces for creating graphic content and rendering that content to the screen.
`CoreText.framework`	Contains the interfaces for performing text layout and display. Available in Mac OS X v10.5 and later.
`HIServices.framework`	Contains interfaces for accessibility, Internet Config, the pasteboard, the Process Manager, and the Translation Manager. Available in Mac OS X 10.2 and later.
`ImageIO.framework`	Contains interfaces for importing and exporting image data. Prior to Mac OS X v10.5, these interfaces were part of the CoreGraphics subframework.
`LangAnalysis.framework`	Contains the Language Analysis Manager interfaces.
`PrintCore.framework`	Contains the Core Printing Manager interfaces.
`QD.framework`	Contains the QuickDraw interfaces.
`SpeechSynthesis.framework`	Contains the Speech Manager interfaces.

Automator Framework

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

Table B-4 Subframeworks of the Automator framework

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

Carbon Framework

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

Table B-5 Subframeworks of the Carbon framework

Subframework	Description
`CarbonSound.framework`	Contains the Sound Manager interfaces. Whenever possible, use Core Audio instead.
`CommonPanels.framework`	Contains interfaces for displaying the Font window, Color window, and some network-related dialogs.
`Help.framework`	Contains interfaces for launching and searching Apple Help.
`HIToolbox.framework`	Contains interfaces for the Carbon Event Manager, HIToolbox object, and other user interface–related managers.
`HTMLRendering.framework`	Contains interfaces for rendering HTML content. For Mac OS X version 10.2 and later, the WebKit framework is the preferred framework for HTML rendering. See "WebKit Framework" (page 125).
`ImageCapture.framework`	Contains interfaces for capturing images from digital cameras. This framework works in conjunction with the Image Capture Devices framework (`ICADevices.framework`).
`Ink.framework`	Contains interfaces for managing pen-based input. (Ink events are defined with the Carbon Event Manager.) Available in Mac OS X version 10.3 and later.
`Navigation-Services.framework`	Contains interfaces for displaying file navigation dialogs.
`OpenScripting.framework`	Contains interfaces for writing scripting components and interacting with those components to manipulate and execute scripts.
`Print.framework`	Contains the Carbon Printing Manager interfaces for displaying printing dialogs and extensions.
`SecurityHI.framework`	Contains interfaces for displaying security-related dialogs.
`Speech-Recognition.framework`	Contains the Speech Recognition Manager interfaces.

Core Services Framework

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

Table B-6 Subframeworks of the Core Services framework

Subframework	Description
`AE.framework`	Contains interfaces for creating and manipulating Apple events and making applications scriptable.
`CarbonCore.framework`	Contains interfaces for many legacy Carbon Managers. In Mac OS X v10.5 and later, this subframework contains the FSEvents API, which notifies clients about file system changes.
`CFNetwork.framework`	Contains interfaces for network communication using HTTP, sockets, and Bonjour.
`DictionaryServices.framework`	Provides dictionary lookup capabilities.
`LaunchServices.framework`	Contains interfaces for launching applications.
`Metadata.framework`	Contains interfaces for managing Spotlight metadata. Available in Mac OS X v10.4 and later.
`OSServices.framework`	Contains interfaces for Open Transport and many hardware-related legacy Carbon managers.
`SearchKit.framework`	Contains interfaces for the Search Kit. Available in Mac OS X version 10.3 and later.

IMCore Framework

Table B-8 lists the subframeworks of the IMCore framework (`IMCore.framework`). This framework was introduced in Mac OS X version 10.6.

Table B-7 Subframeworks of the IMCore framework

Subframework	Description
`IMDaemonCore.framework`	Contains private interfaces.
`IMFoundation.framework`	Contains private interfaces.
`IMSecurityUtils.framework`	Contains private interfaces.
`IMUtils.framework`	Contains private interfaces.
`XMPPCore.framework`	Contains private interfaces.

Quartz Framework

Table B-8 lists the subframeworks of the Quartz framework (`Quartz.framework`). This framework was introduced in Mac OS X version 10.4.

Table B-8 Subframeworks of the Quartz framework

Subframework	Description
`ImageKit.framework`	Contains Objective-C interfaces for finding, browsing, and displaying images. Available in Mac OS X version 10.5 and later.
`PDFKit.framework`	Contains Objective-C interfaces for displaying and managing PDF content in windows.
`QuartzComposer.framework`	Contains Objective-C interfaces for playing Quartz Composer compositions in an application.
`QuartzFilters.framework`	Contains Objective-C interfaces for managing and applying filter effects to a graphics context. Available in Mac OS X version 10.5 and later.

WebKit Framework

Table B-9 lists the subframeworks of the WebKit framework (`WebKit.framework`). This framework was introduced in Mac OS X version 10.2.

Table B-9 Subframeworks of the WebKit framework

Subframework	Description
`WebCore.framework`	Contains the library and resources for rendering HTML content in an HTMLView control.

Xcode Frameworks

In Mac OS X v10.5 and later, Xcode and all of its supporting tools and libraries reside in a portable directory structure. This directory structure makes it possible to have multiple versions of Xcode installed on a single system or to have Xcode installed on a portable hard drive that you plug in to your computer when you need to do development. This portability means that the frameworks required by the developer tools are installed in the `<Xcode>/Library/Frameworks` directory, where `<Xcode>` is the path to the Xcode installation directory. (The default Xcode installation directory is `/Developer`.) Table B-10 lists the frameworks that are located in this directory.

Table B-10 Xcode frameworks

Framework	First available	Prefixes	Description
`CPlusTest.framework`	10.4	None	Unit-testing framework for C++ code. In Mac OS X v10.4, this framework was in `/System/Library/Frameworks`.
`InterfaceBuilder-Kit.framework`	10.5	`ib`, `IB`	Contains interfaces for writing plug-ins that work in Interface Builder v3.0 and later.

Framework	First available	Prefixes	Description
`SenTesting-Kit.framework`	10.4	`Sen`	Contains the interfaces for implementing unit tests in Objective-C. In Mac OS X v10.4, this framework was in `/System/Library/Frameworks`.

System Libraries

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

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

Mac OS X Developer Tools

Apple provides a number of applications and command-line tools to help you develop your software. These tools include compilers, debuggers, performance analysis tools, visual design tools, scripting tools, version control tools, and many others. Many of these tools are installed with Mac OS X by default but the rest require you to install Xcode first. Xcode is available for free from the Apple Developer Connection website. For more information on how to get these tools, see "Getting the Xcode Tools" (page 14).

Note: Documentation for most of the command-line tools is available in the form of `man` pages. You can access these pages from the command line or from *Mac OS X Man Pages*. For more information about using the command-line tools, see "Command Line Primer" (page 109).

Applications

Xcode includes numerous applications for writing code, creating resources, tuning your application, and delivering it to customers. At the heart of this group is the Xcode application, which most developers use on a daily basis. It provides the basic project and code management facilities used to create most types of software on Mac OS X. All of the tools are free and can be downloaded from the Apple developer website (see "Getting the Xcode Tools" (page 14)).

In Mac OS X v10.5 and later, it is possible to install multiple versions of Xcode on a single computer and run the applications and tools from different versions side-by-side. The applications listed in the following sections are installed in <*Xcode*>`/Applications`, where <*Xcode*> is the root directory of your Xcode installation. The default installation directory for Xcode is the `/Developer` directory.

In addition to the applications listed here, Xcode also comes with numerous command-line tools. These tools include the GCC compiler GDB debugger, tuning tools, code management tools, performance tools, and so on. For more information about the available command-line tools, see "Command-Line Tools" (page 143).

Xcode

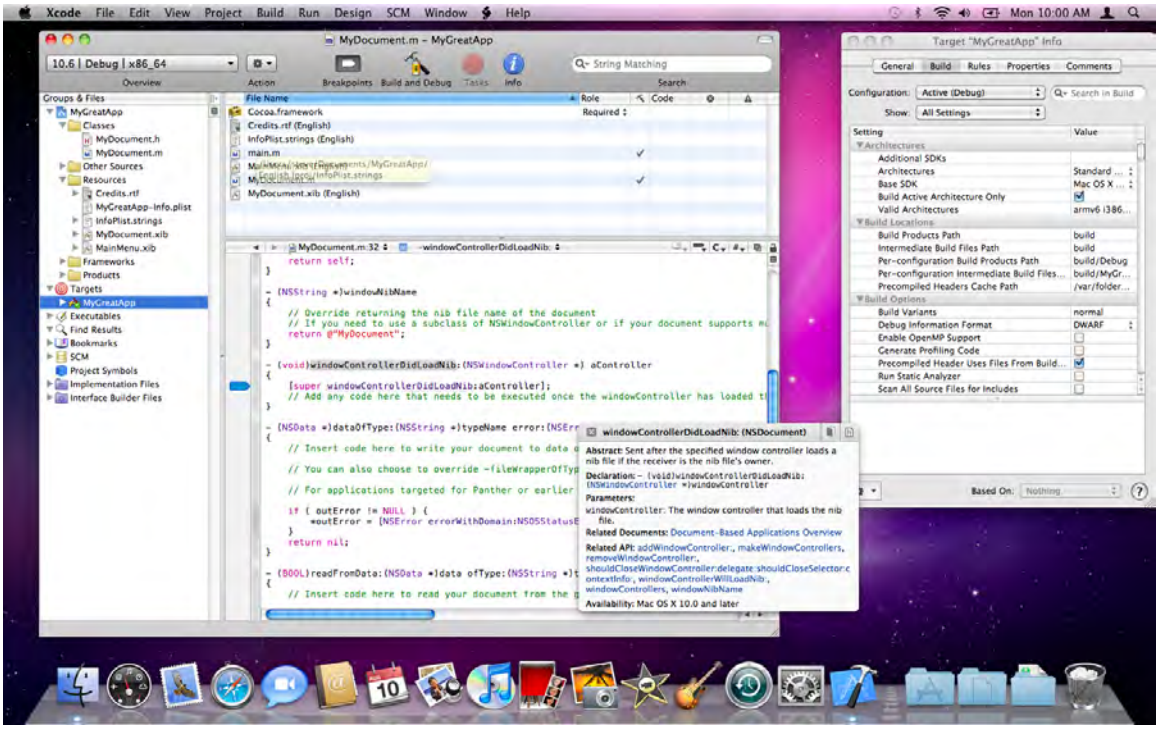
The centerpiece of the Xcode Tools is the Xcode application, which is an integrated developer environment (IDE) with the following features:

- A project management system for defining software products
- A code editing environment that includes features such as syntax coloring, code completion, and symbol indexing; see"Xcode Editor" (page 129)
- Visual design tools for creating your application's data model (see "Core Data Framework" (page 62))
- An advanced documentation viewer for viewing and searching Apple documentation; see "Documentation Window" (page 130)

- A context-sensitive inspector for viewing information about selected code symbols; see "Research Assistant" (page 130)
- An advanced build system with dependency checking and build rule evaluation.
- GCC compilers supporting C, C++, Objective-C, Objective-C++, Objective-C 2.0, and other compilers supporting Java and other languages
- Integrated source-level debugging using GDB; see "Debugging Environment" (page 130)
- Distributed computing, enabling you to distribute large projects over several networked machines
- Predictive compilation that speeds single-file compile turnaround times
- Advanced debugging features such as fix and continue and custom data formatters
- Advanced refactoring tools that let you make global modifications to your code without changing its overall behavior; see "Refactoring Tools" (page 132)
- Support for project snapshots, which provide a lightweight form of local source code management; see "Project Snapshots" (page 132)
- Support for launching performance tools to analyze your software
- Support for integrated source-code management; see "SCM Repository Management" (page 131)
- AppleScript support for automating the build process
- Support for the ANT build system, which can be used to build Java and WebObjects projects.
- Support for DWARF and Stabs debugging information (DWARF debugging information is generated by default for all new projects)

Figure C-1 shows the Xcode project workspace and some key inspector windows. In the Xcode preferences, you can configure numerous aspects of the workspace to suit your preferred work style.

Figure C-1 Xcode application



For an introduction to Xcode and its features, see *A Tour of Xcode*.

Xcode Editor

The Xcode editing environment is a high-performance code editor that includes many features that go beyond basic text editing. These features aim to help developers create better code faster and include the following:

- High-performance for typing, scrolling, and opening files. The Xcode editor now opens and scrolls large source documents up to 10 times faster than before.
- Code annotations display notes, errors, and warnings inline with the code itself, and not just as icons in the gutter. This provides a much more direct conveyance of where the problems in your code lie. You can control the visibility of annotations using the segmented control in the navigation bar.
- Code folding helps you organize your source files by letting you temporarily hide the content of a method or function in the editor window. You can initiate code folding by holding down the Command and Option keys and pressing either the left or right arrow key. A ribbon to the left of the text shows the current nesting depth and contains widgets to fold and unfold code blocks.
- Syntax coloring lets you assign colors to various code elements, including keywords, comments, variables, strings, class names, and more.
- Code Sense code completion, a feature that shows you type a few characters and retrieve a list of valid symbol names that match those characters. Code Sense is fast and intuitive and is tuned to provide accurate completions, along with a “most likely” inline completion as you type. This feature is similar to the auto-completion features found in Mail, Terminal, and other applications.

Debugging Environment

In Xcode 3.0, there is no distinction between "Running" your executable and "Debugging" it. Instead, you simply build your executable and run it. Hitting a breakpoint interrupts the program and displays the breakpoint either in the current editor window or in the debugger window. Other features of the debugging environment include the following:

- Debugging controls in editor windows.
- A debugger HUD (heads-up-display), which is a floating window with debugger controls that simplifies the debugging of full-screen applications.
- Variable tooltips. (Moving your mouse over any variable displays that variable's value.)
- Reorganization (and in some cases consolidation) of toolbar and menu items to improve space usage, while still keeping all the needed tools available.
- Consolidation of the Standard I/O Log, Run Log, and Console log into the Console log window.
- Support for a separate debugging window if you prefer to debug your code that way.

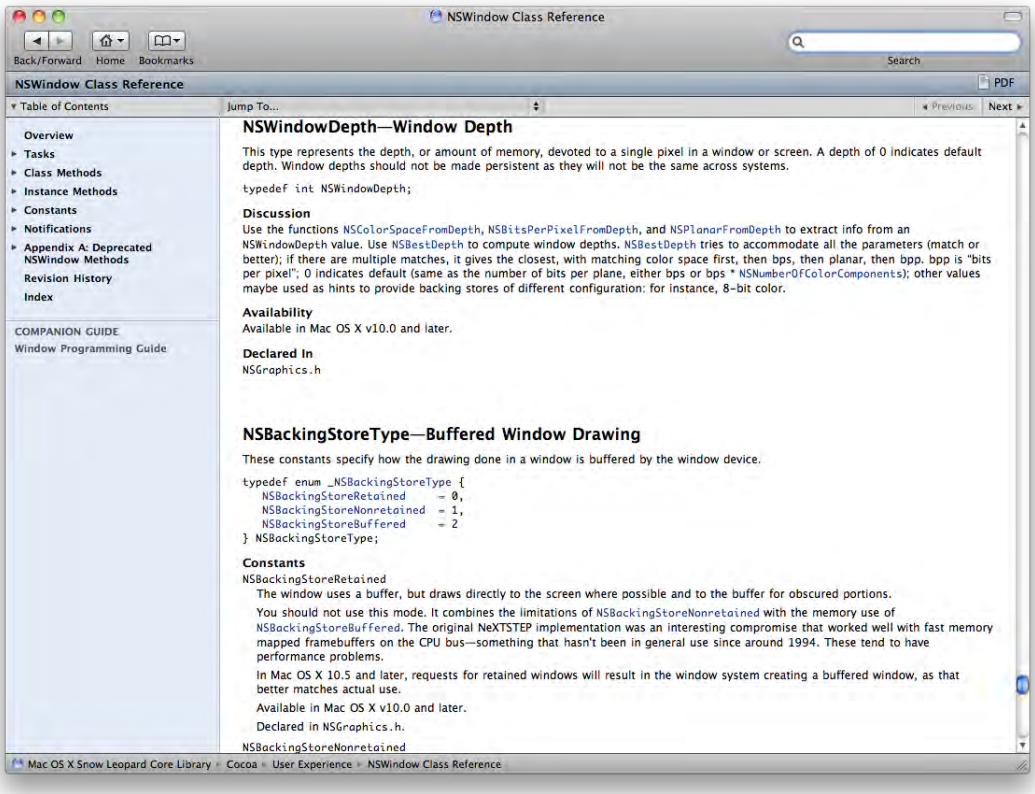
Research Assistant

The Research Assistant is an inspector that displays documentation for the currently selected text (see Figure C-1 (page 129)). As the selection changes, the Research Assistant updates the information in its floating window to reflect the classes, methods, and functions you are currently using. This window shows the declaration, abstract, and availability information for the selection along with the framework containing the selected identifier, relevant documentation resources, and related methods and functions you might be interested in using.

Documentation Window

The documentation window (Figure C-2) in Xcode provides an environment for searching and browsing the documentation. This window provides you with fast access to Apple's developer documentation and gives you tools for searching its content. You can search by title, by language, and by content and can focus your search on the documents in a particular documentation set.

Figure C-2 Xcode documentation window



Documentation sets are collections of documents that can be installed, browsed, and searched independently. Documentation sets make it easier to install only the documentation you need for your development, reducing the amount of disk space needed for your developer tools installation. In addition to the Apple-provided documentation sets, third parties can implement their own documentation sets and have them appear in the Xcode documentation window. For information on how to create custom documentation sets, see *Documentation Set Guide*.

SCM Repository Management

Xcode supports the management of multiple SCM repositories to allow you to perform tasks such as the following:

- Initial checkout of projects
- Tagging source files
- Branching
- Importing and exporting files

Xcode supports CVS, Subversion, and Perforce repositories.

Project Snapshots

Project snapshots provide a lightweight form of local source control for Xcode projects. Using this feature, you can take a "snapshot" of your project's state at any point during development, such as after a successful build or immediately prior to refactoring your code. If after making subsequent changes you decide those changes were not useful, you can revert your project files back to the previous snapshot state. Because snapshots are local, your intermediate changes need never be committed to source control.

Refactoring Tools

Xcode's refactoring tools let you make large-scale changes to your Objective-C source code quickly and easily. Xcode propagates your change requests throughout your code base, making sure that the changes do not break your builds. You can make the following types of changes using the refactoring tools:

- Rename instance methods
- Create new superclasses
- Move methods into a superclass
- Convert accessor methods to support Objective-C 2.0 properties
- Modernize appropriate `for` loops to use the new fast enumeration syntax introduced in Objective-C 2.0

Before making any changes to your code, Xcode's refactoring tools automatically take a local snapshot of your project. This automatic snapshot means you can experiment with refactoring changes without worrying about irrevocably changing your project files. For more information on snapshots, see "Project Snapshots" (page 132).

Build Settings

The Build pane in the inspector organizes the build settings for the selected target, providing search tools to help you find particular settings. In Mac OS X v10.5, some particularly noteworthy additions to this pane include the following:

- Per-architecture build settings. You can now set different build settings for each architecture (Intel, PowerPC) your product supports.
- 32-bit and 64-bit architecture checkboxes.

Project Versioning

Xcode projects include a Compatibility pane in the project inspector that lets you determine whether you want an Xcode 3.0–only project or one that can be used by previous versions of Xcode. Marking a project as Xcode 3.0–only generates an alert whenever you try to use an Xcode feature that is not present in previous versions of the application.

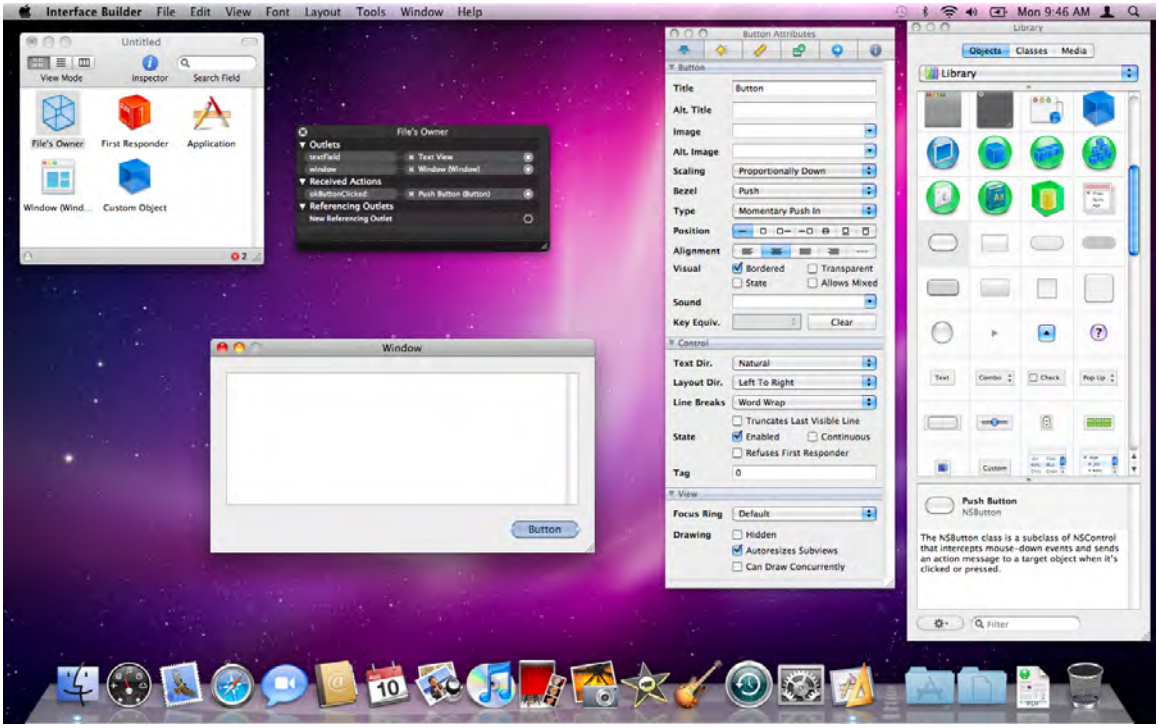
Interface Builder

The Interface Builder application provides a graphical environment for building the user interface of your Carbon and Cocoa applications. Using Interface Builder, you assemble the windows and menus of your application, along with the any other supporting objects, and save them in one or more resource files, called nib files. When you want to load a user interface element at runtime, you load the nib file. The Cocoa and Carbon infrastructure uses the data in the nib file to recreate the objects exactly as they were in Interface Builder, with all their attributes and inter-object relationships restored.

Although present in all versions of Mac OS X, the Interface Builder application received a significant overhaul in Mac OS X v10.5. Beyond the numerous cosmetic changes, the current version of Interface Builder includes numerous workflow and infrastructure changes too. The connections panel replaces the old technique for connecting objects in Cocoa nib files, making it possible to create multiple connections quickly without going back and forth between the inspector and the objects in your nib file. An improved library window helps you organize and find the components you use most frequently. Interface Builder includes a new plug-in model that makes it possible to create fully functional plug-ins in a matter of minutes. And most importantly, Interface Builder is more tightly integrated with Xcode, providing automatic synchronization of project's class information with the corresponding source files.

Figure C-3 shows the Interface Builder environment in Mac OS X v10.5, including a nib document, connections panel, inspector window, and library window. The library window contains the standard components you use to build your user interfaces and includes all of the standard controls found in Carbon and Cocoa applications by default. Using plug-ins, you can expand the library to include your own custom objects or to include custom configurations of standard controls.

Figure C-3 Interface Builder 3.0



For information about Interface Builder features and how to use them, see *Interface Builder User Guide*. For information about how to integrate your own custom controls into Interface Builder, see *Interface Builder Plug-In Programming Guide* and *Interface Builder Kit Framework Reference*.

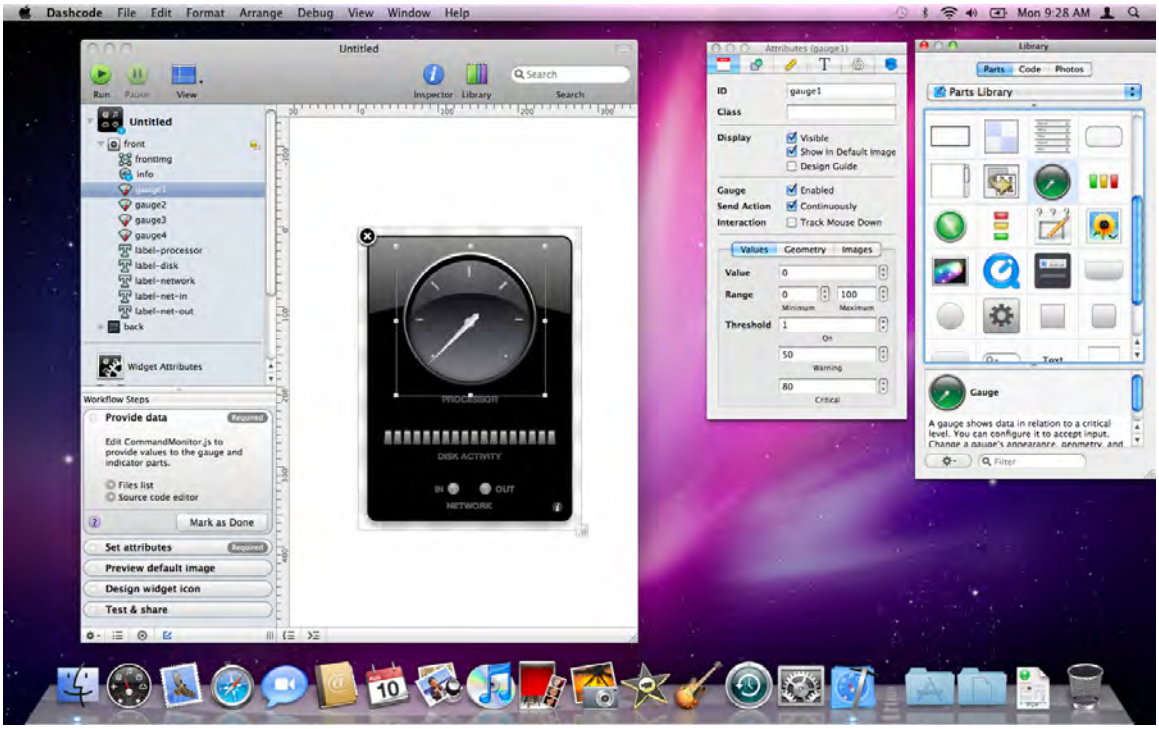
Dashcode

Introduced in Mac OS X v10.5, Dashcode is an integrated environment for laying out, coding, and testing Dashboard widgets. Although users see and use widgets as applications, they're actually packaged webpages powered by standard technologies such as HTML, CSS, and JavaScript. Although it is easy for anyone with web design experience to build a widget using existing webpage editors, as a widget's code and layout get more complex, managing and testing of that widget becomes increasingly difficult. Dashcode provides the following features to help simplify the widget design process:

- A project manager to marshall your widget's resources
- Visual tools to design your widget interface
- Tools to set metadata values, specify required images, and package your widget
- A source code editor to implement your widget's behavior
- A debugger to help you resolve issues in your widget's implementation

Figure C-4 shows the Dashcode canvas, inspector, and library windows. The canvas is a drag-and-drop layout environment where you lay out widgets visually. Using the inspector window, you can apply style information to the controls, text, and shape elements that you drag in from the library.

Figure C-4 Dashcode canvas



For more information about Dashcode, see *Dashcode User Guide*.

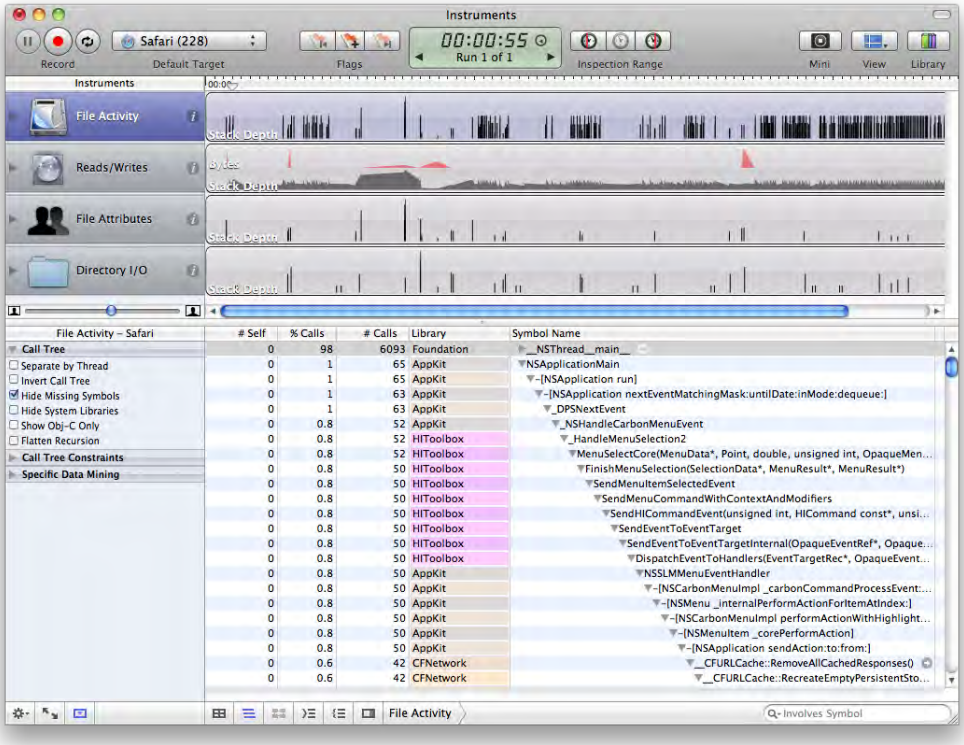
Instruments

Introduced in Mac OS X v10.5, Instruments is an advanced debugging and performance analysis application. Instruments provides unprecedented information about the runtime behavior of your application and complements existing tools such as Shark. Rather than show one aspect of your program at a time, you configure each analysis session with one or more "instruments", each of which gathers information about things such as object allocation patterns, memory usage, disk I/O, CPU usage, and many more. The data from all instruments is shown side-by-side, making it easier to see patterns between different types of information.

An important aspect of Instruments is the repeatability of data gathering operations. Instruments lets you record a sequence of events in your application and store them in the master track. You can then replay that sequence to reproduce the exact same conditions in your application. This repeatability means that each new set of data you gather can be compared directly to any old sets, resulting in a more meaningful comparison of performance data. It also means that you can automate much of the data gathering operation. Because events are shown alongside data results, it is easier to correlate performance problems with the events that caused them.

Figure C-5 shows the Instruments user interface for an existing session. Data for each instrument is displayed along the horizontal axis. Clicking in those data sets shows you information about the state of the application at that point in time.

Figure C-5 The Instruments application interface

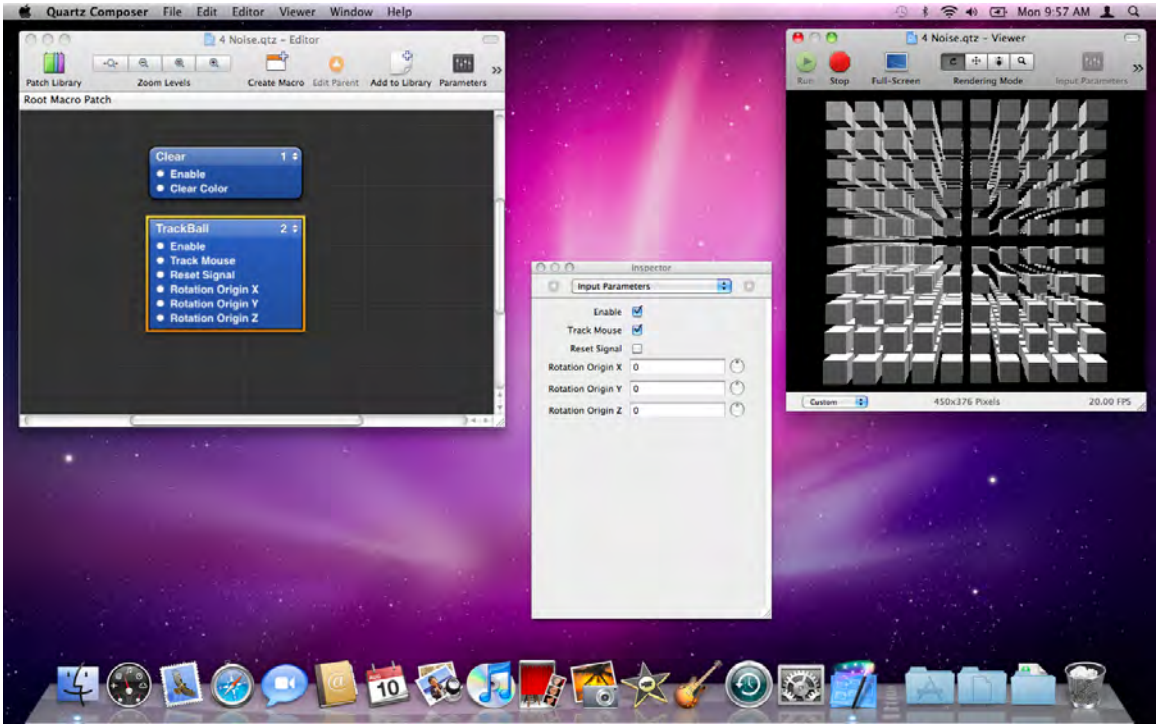


For information about how to use Instruments, see *Instruments User Guide*.

Quartz Composer

Introduced in Mac OS X version 10.4, Quartz Composer is a development tool for processing and rendering graphical data. Quartz Composer provides a visual development environment (Figure C-6) built on technologies such as Quartz 2D, Core Image, OpenGL, and QuickTime. You can use Quartz Composer as an exploratory tool to learn the tasks common to each visual technology without having to learn its application programming interface (API). You can also save your compositions as resource files that can be loaded into a Cocoa window at runtime. In addition to supporting visual technologies, Quartz Composer also supports nongraphical technologies such as MIDI System Services and Rich Site Summary (RSS) file content.

Figure C-6 Quartz Composer editor window



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

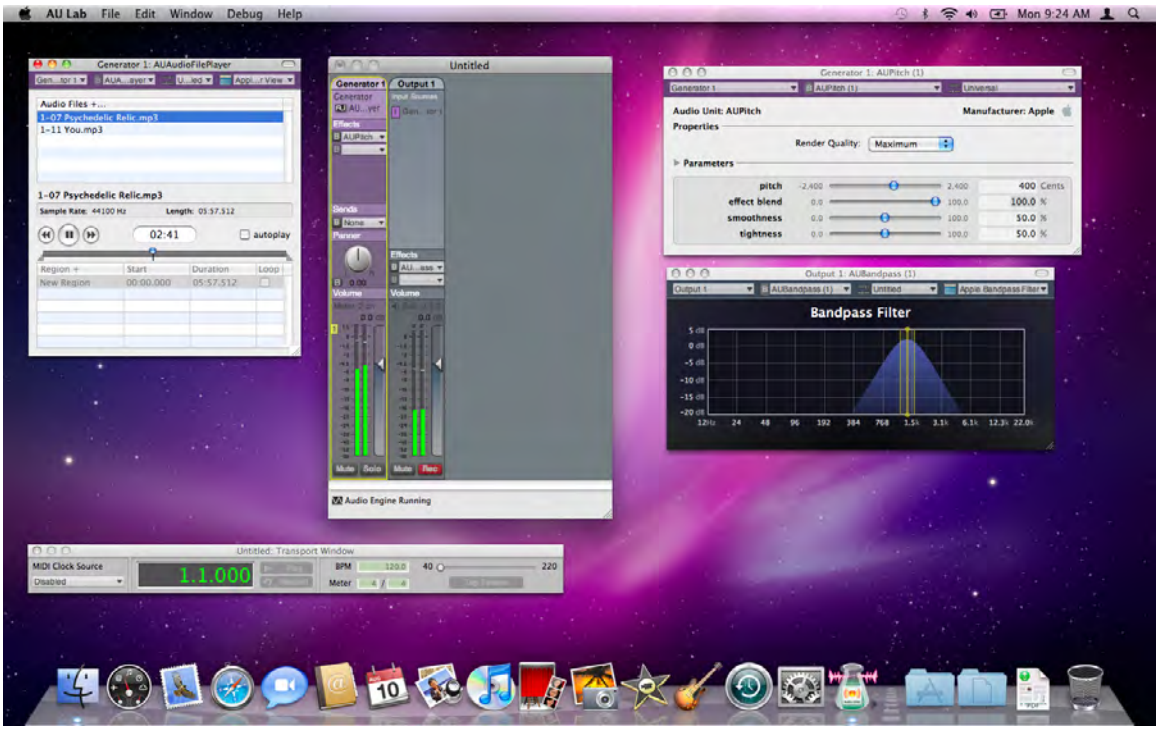
Audio Applications

The <*Xcode*>`/Applications/Audio` directory contains applications for Core Audio developers.

AU Lab

Introduced in Mac OS X version 10.4, AU Lab (Audio Unit Lab) lets you graphically host audio units and examine the results. You can use AU Lab to test the audio units you develop, do live mixing, and playback audio content. Parameters for the audio units are controlled graphically using the audio unit's custom interface or using a generic interface derived from the audio unit definition. Figure C-7 shows the AU Lab interface and some of the palettes for adjusting the audio parameters.

Figure C-7 AU Lab mixer and palettes



HALLab

Introduced in Mac OS X version 10.5, the HALLab (Hardware Abstraction Layer Lab) application helps developers test and debug audio hardware and drivers. You can use this application to understand what the audio hardware is doing and to correlate the behavior of your application with the behavior of the underlying audio driver. The application provides information about the properties of objects in the HAL and provides an I/O cycle telemetry viewer for diagnosing and debugging glitches your application's audio content.

Graphics Applications

Table C-1 lists the applications found in the <*Xcode*>`/Applications/Graphics Tools` directory.

Table C-1 Graphics applications

Application	Description
Core Image Fun House	Introduced in Mac OS X v10.5, this application provides an environment for testing the effects of Core Image filters. Using this application, you can build up a set of filters and apply them to an image or set of images. You can apply both static and time-based effects and change the parameters of each filter dynamically to see the results.
OpenGL Driver Monitor	An application that displays extensive information about the OpenGL environment.
OpenGL Profiler	An application that creates a runtime profile of an OpenGL-based application. The profile contains OpenGL function-call timing information, a listing of all the OpenGL function calls your application made, and all the OpenGL-related data needed to replay your profiling session.
OpenGL Shader Builder	An application that provides real-time entry, syntax checking, debugging, and analysis of vertex/fragment programs. It allows you to export your creation to a sample GLUT application, which performs all the necessary OpenGL setup, giving you a foundation to continue your application development. OpenGL is an open, cross-platform, three-dimensional (3D) graphics standard that supports the abstraction of current and future hardware accelerators. For more information about OpenGL, see *OpenGL Programming Guide for Mac OS X* in the Reference Library > Graphics & Imaging area.
Pixie	A magnifying glass utility for Mac OS X. Pixie is useful for doing pixel-perfect layout, checking the correctness of graphics and user interface elements, and getting magnified screen shots.
Quartz Composer Visualizer	A utility for previewing Quartz Composer compositions.
Quartz Debug	This is an alias to the Quartz Debug application in the *<Xcode>*`/Applications/Performance Tools` directory. For more information, see the entry for Quartz Debug in "Performance Applications" (page 139).

Java

Table C-2 lists the applications found in the *<Xcode>*`/Applications/Java Tools` directory.

Table C-2 Java applications

Application	Description
Applet Launcher	An application that acts as a wrapper for running Java applets.
Jar Bundler	An application that allows you to package your Java program's files and resources into a single double-clickable application bundle. Jar Bundler lets you modify certain properties so your Java application behaves as a better Mac OS X citizen and lets you specify arguments sent to the Java virtual machine (VM) when the application starts up.

Performance Applications

Table C-3 lists the applications found in the *<Xcode>*`/Applications/Performance Tools` directory.

Table C-3 Performance applications

Application	Description
BigTop	An application that presents statistics about the current system activity and lets you track those statistics over time. This application is a more visually oriented version of the top command-line tool. It provides information about CPU usage, disk and network throughput, memory usage, and others. For information on how to use this program, see the application help.
MallocDebug	An application for measuring the dynamic memory usage of applications and for finding memory leaks. For information on how to use this program, see the application help or *Memory Usage Performance Guidelines*.
Quartz Debug	A debugging utility for the Quartz graphics system. For information on how to use this program, see the application help or *Drawing Performance Guidelines*.
Shark	An application that profiles the system to see how time is being spent. It can work at the system, task, or thread level and can correlate performance counter events with source code. Shark's histogram view can be used to observe scheduling and other time-dependent behavior. It can produce profiles of hardware and software performance events such as cache misses, virtual memory activity, instruction dependency stalls, and so forth. For information on how to use this program, see the application help.
Spin Control	An application that samples applications automatically whenever they become unresponsive and display the spinning cursor. To use this application, you launch it and leave it running. Spin Control provides basic backtrace information while an application is unresponsive, showing you what the application was doing at the time.
Thread Viewer	An application for graphically displaying activity across a range of threads. It provides timeline color-coded views of activity on each thread. By clicking a point on a timeline, you can see a sample backtrace of activity at that time.
ZoneMonitor	An application for analyzing memory usage.

Table C-4 lists the applications in the *<Xcode>*`/Applications/Performance Tools/CHUD` directory and its subdirectories.

Table C-4 CHUD applications

Application	Description
Reggie SE	An application that examines and modifies CPU and PCI configuration registers in PowerPC processors.
PMC Index	An application for finding performance counter events and their configuration.

Application	Description
Saturn	An application that is an exact, function-level profiler for your application. Unlike sampling programs, which gather call stacks at periodic intervals, you can use this application to generate and view a complete function call trace of your application code.
SpindownHD	An application that monitors the power state of hard drives connected to the computer.

Utility Applications

Table C-5 lists the applications found in the *<Xcode>*`/Applications/Graphics Tools` directory and its subdirectories.

Table C-5 Utility applications

Application	Description
Accessibility Inspector	An agent application that lets you roll the mouse cursor over items in your application's user interface and view their associated accessibility attributes and actions.
Accessibility Verifier	An application that looks for mistakes in the accessibility information provided by your application.
Bluetooth Explorer	An application for discovering and getting information about Bluetooth devices.
Build Applet	An application for creating applets from Python scripts.
Clipboard Viewer	An application that displays the contents of the various system pasteboards.
CrashReporterPrefs	An application for configuring the user notifications generated when an application crashes.
FileMerge	An application that compares two ASCII files or two directories. For a more accurate comparison, you can compare two files or directories to a common ancestor. After comparing, you can merge the files or directories.
Help Indexer	An application to create a search index for a help file. Instructions for creating Apple Help and for using the indexing tool are in *Apple Help Programming Guide*.
Icon Composer	An application for creating and examining icon resource files.
IORegistryExplorer	An application that you can use to examine the configuration of devices on your computer. IORegistryExplorer provides a graphical representation of the I/O Registry tree. For information on how to use this application, see *I/O Kit Fundamentals*.
iSync Plug-in Maker	This application creates device drivers that allow the synchronization of custom hardware devices. For more information, see "iSync Plug-in Maker" (page 141).
PackageMaker	An application for creating installable application packages from the set of files you provide.
PacketLogger	An application for logging Bluetooth packets.

Application	Description
Property List Editor	An application that lets you read and edit the contents of a property list. A property list, or plist, is a data representation used by Cocoa and Core Foundation as a convenient way to store, organize, and access standard object types. Property lists are useful when you need to store small amounts of persistent data. If you do use property lists, the `.plist` files must be bundled with other application files when you create your installation package.
Repeat After Me	An application designed to improve the pronunciation of text generated by the Text-To-Speech system.
SRLanguageModeler	An application for building language models for use with the Speech Recognition manager.
Syncrospector	A debugging application you use to inspect the truth database, the call history of sync sessions, and clients of the synchronization engine. For information on how to use this application, see *Sync Services Tutorial*.
USB Prober	An application that displays detailed information about all the USB ports and devices on the system.

iSync Plug-in Maker

The iSync Plug-in Maker application is a tool that allows you to build, test, and release plug-ins that handle the specific features supported by your hardware device. You use this application to configure your device settings and write scripts for connecting it to the Internet. The application also provides a suite of standard automated tests that you can use to detect and fix problems in your plug-in before you ship it.

Figure C-8 shows the iSync Plug-in Maker edit window.

Figure C-8 iSync Plug-in Maker application

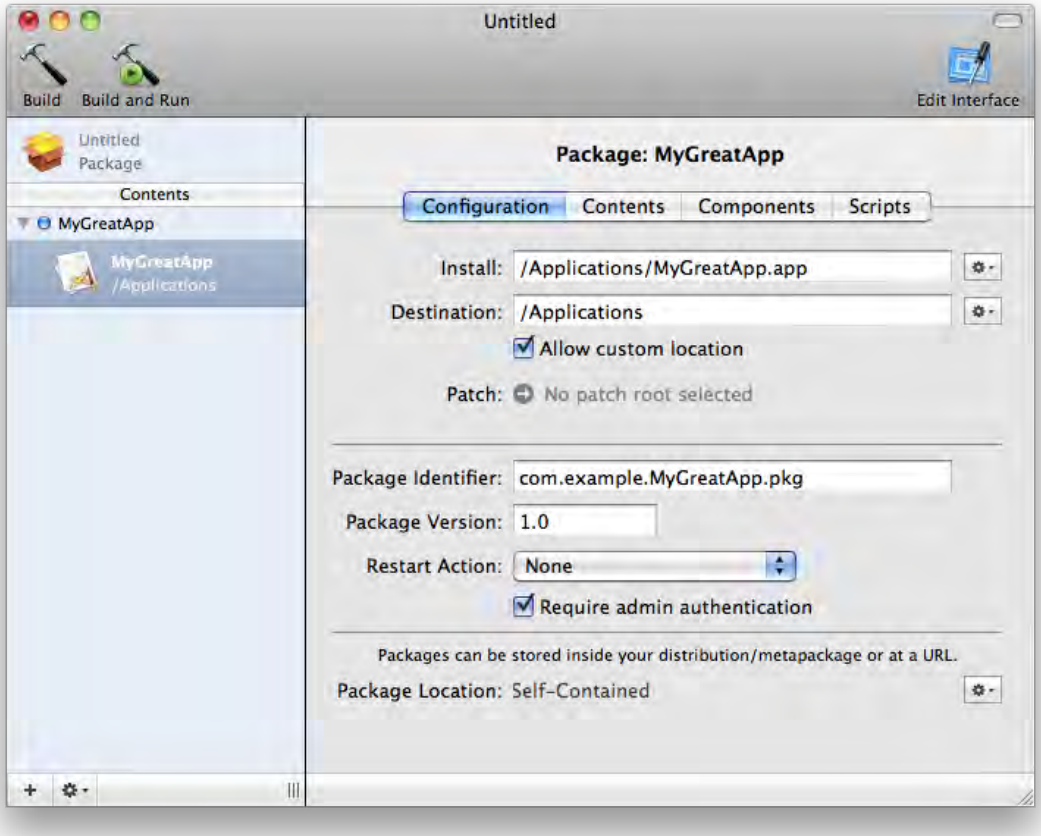
For information about using iSync Plug-in Maker, see *iSync Plug-in Maker User Guide*.

PackageMaker

You use PackageMaker to create installation packages for your software. An installation package is a convenient way to deliver software in all but the simplest cases. An installation package contains the files to install, their locations, and any licensing information or supporting materials that go with your software. When the user double-clicks an installation package, Mac OS X automatically launches the Installer application, which proceeds to install the files contained in the package.

You can use PackageMaker to package files or to assemble individual packages into a single package. Figure C-9 shows the PackageMaker user interface, which provides a graphical environment for building your packages.

Figure C-9 PackageMaker application



For information on how to use PackageMaker, see *PackageMaker User Guide*.

Command-Line Tools

Xcode includes numerous command-line tools, including the GCC compiler, GDB debugger, performance tools, version control system tools, localization tools, scripting tools, and so on. Some of these tools are found on most other BSD-based installations while others were created specifically for Mac OS X. They come free with the rest of Xcode, which you can download from the Apple developer website (see "Getting the Xcode Tools" (page 14)).

In Mac OS X v10.5 and later, it is possible to install multiple versions of Xcode on a single computer and run the applications and tools from different versions side-by-side. Most of the tools listed in the following sections are installed in either in the system's `/usr/bin` directory or in *<Xcode>*`/usr/bin` or *<Xcode>*`/usr/sbin`, where *<Xcode>* is the root directory of your Xcode installation, although tools installed elsewhere are called out as such. The default installation directory for Xcode is the `/Developer` directory.

In addition to the command-line tools listed here, Xcode also comes with many higher-level applications. These tools include the Xcode integrated development environment, Interface Builder, Instruments, and many others. For more information about the available applications, see "Applications" (page 127).

Note: The following sections describe some of the more important tools provided with Xcode but should by no means be considered a complete list. If the tool you are looking for is not described here, check the system and Xcode tools directories or see *Mac OS X Man Pages*.

Compiler, Linker, and Source Code Tools

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

Compilers, Linkers, Build Tools

Table C-6 lists the command-line compilers, linkers, and build tools. These tools are located in *<Xcode>*`/usr/bin` and *<Xcode>*`/Private`.

Table C-6 Compilers, linkers, and build tools

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

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

Library Utilities

Table C-7 lists the command-line tools available for creating libraries. These tools are located in *<Xcode>*`/usr/bin`.

Table C-7 Tools for creating and updating libraries

Tool	Description
`libtool`	Takes object files and creates dynamically linked libraries or archive (statically linked) libraries, according to the options selected. The `libtool` command calls the `ld` command. See `libtool` man page.
`lorder`	Determines interdependencies in a list of object files. The output is normally used to determine the optimum ordering of the object modules when a library is created so that all references can be resolved in a single pass of the loader. See `lorder` man page.
`ranlib`	Adds to or updates the table of contents of an archive library. See `ranlib` man page.
`redo_prebinding`	Updates the prebinding of an executable or dynamic library when one of the dependent dynamic libraries changes. (Prebinding for user applications is unnecessary in Mac OS X v10.3.4 and later.) See `redo_prebinding` man page.
`update_prebinding`	Updates prebinding information for libraries and executables when new files are added to the system. (Prebinding for user applications is unnecessary in Mac OS X v10.3.4 and later.) See `update_prebinding` man page.

Code Utilities

Table C-8 lists applications and command-line tools for manipulating source code and application resources. These tools are located in *<Xcode>*`/usr/bin` and `/usr/bin`.

Table C-8 Code utilities

Tool	Description
`ifnames`	Scans C source files and writes out a sorted list of all the identifiers that appear in `#if`, `#elif`, `#ifdef`, and `#ifndef` directives. See `ifnames` man page.
`indent`	Formats C source code. See `indent` man page.

Tool	Description
nmedit	Changes global symbols in object code to static symbols. You can provide an input file that specifies which global symbols should remain global. The resulting object can still be used with the debugger. See nmedit man page.
plutil	Can check the syntax of a property list or convert it from one format to another (XML or binary). See plutil man page.
printf	Formats and prints character strings and C constants. See printf man page.
ResMerger	Merges resources into resource files. When the Xcode application compiles Resource Manager resources, it sends them to a collector. After all Resource Manager resources have been compiled, the Xcode application calls ResMerger to put the resources in their final location. See ResMerger man page.
RezWack	Takes a compiled resource (.qtr) file and inserts it together with the data fork (.qtx or .exe file) into a Windows application (.exe) file. The resulting file is a Windows application that has the sort of resource fork that QuickTime understands. You can use the Rez tool to compile a resource source (.r) file. The RezWack tool is part of the QuickTime 3 Software Development Kit for Windows. See RezWack man page.
tops	Performs universal search and replace operations on text strings in source files. See tops man page.
unifdef	Removes #ifdef, #ifndef, #else, and #endif lines from code as specified in the input options. See unifdef man page.
UnRezWack	Reverses the effects of RezWack; that is, converts a single Windows executable file into separate data and resource files. See UnRezWack man page.

Debugging and Tuning Tools

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

General Tools

Table C-9 lists the command-line tools available for debugging. These tools are located in *<Xcode>*/usr/bin and /usr/bin.

Table C-9 General debugging tools

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

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

Memory Analysis Tools

Table C-10 lists the applications and command-line tools for debugging and tuning memory problems. These tools are located in <*Xcode*>/usr/bin and /usr/bin.

Table C-10 Memory debugging and tuning tools

Tool	Description
heap	Lists all the objects currently allocated on the heap of the current process. It also describes any Objective-C objects, listed by class. See heap man page.
leaks	Examines a specified process for malloc-allocated buffers that are not referenced by the program. See leaks man page.
malloc_history	Inspects a given process and lists the malloc allocations performed by it. This tool relies on information provided by the standard malloc library when debugging options have been turned on. If you specify an address, malloc_history lists the allocations and deallocations that have manipulated a buffer at that address. For each allocation, a stack trace describing who called malloc or free is listed. See malloc_history man page.
vmmap	Displays the virtual memory regions allocated in a specified process, helping you understand how memory is being used and the purpose of memory (text segment, data segment, and so on) at a given address. See vmmap man page.
vm_stat	Displays Mach virtual memory statistics. See vm_stat man page.

Examining Code

Table C-11 lists the applications and command-line tools for examining generated code files. These tools are located in <*Xcode*>/usr/bin and /usr/bin.

Table C-11 Tools for examining code

Tool	Description
c2ph	Parses C code and outputs debugger information in the Stabs format, showing offsets of all the members of structures. For information on Stabs, see *STABS Debug Format*. See c2ph man page.
cscope	An interactive command-line tool that allows the user to browse through C source files for specified elements of code, such as functions, function calls, macros, variables, and preprocessor symbols. See cscope man page.

Tool	Description
ctags	Makes a tags file for the ex line editor from specified C, Pascal, Fortran, YACC, Lex, or Lisp source files. A tags file lists the locations of symbols such as subroutines, typedefs, structs, enums, unions, and #defines. See ctags man page.
error	Analyzes error messages and can open a text editor to display the source of the error. The error tool is run with its input connected via a pipe to the output of the compiler or language processor generating the error messages. Note that the service provided by the error command is built into the Xcode application. See error man page.
ibtool	Lets you print, update, and verify the contents of a nib file. You can use this tool to inject localized strings into a nib file or scan the contents of a nibfile using a script. (This tool replaces the nibtool program.) See ibtool man page.
nm	Displays the symbol tables of one or more object files, including the symbol type and value for each symbol. See nm man page.
otool	Displays specified parts of object files or libraries. See otool man page.
pagestuff	Displays information about the specified logical pages of a file conforming to the Mach-O executable format. For each specified page of code, pagestuff displays symbols (function and static data structure names). See pagestuff man page.
pstruct	An alias to c2ph. See pstruct man page.
strings	Looks for ASCII strings in an object file or other binary file. See strings man page.

Performance Tools

Table C-12 lists the applications and command-line tools for analyzing and monitoring performance. For information about performance and the available performance tools, see *Performance Overview*. These tools are located in *<Xcode>*/usr/bin and /usr/bin.

Table C-12 Performance tools

Tool	Description
gprof	Produces an execution profile of a C, Pascal, or Fortran77 program. The tool lists the total execution times and call counts for each of the functions in the application, and sorts the functions according to the time they represent including the time of their call graph descendants. See gprof man page.
sample	Gathers data about the running behavior of a process. The sample tool stops the process at user-defined intervals, records the current function being executed by the process, and checks the stack to find how the current function was called. It then lets the application continue. At the end of a sampling session, sample produces a report showing which functions were executing during the session. See sample man page.
top	Displays an ongoing sample of system-use statistics. It can operate in various modes, but by default shows CPU and memory use for each process in the system. See top man page.

Instruction Trace Tools

Table C-13 lists the applications and command-line tools for working with hardware-level programs. These tools are located in `/usr/bin`.

Table C-13 Instruction trace tools

Tool	Description
`acid`	Analyzes TT6E (but not TT6) instruction traces and presents detailed analyses and histogram reports. See `acid` man page.
`amber`	Captures the instruction and data address stream generated by a process running in Mac OS X and saves it to disk in TT6, TT6E, or FULL format. Custom trace filters can be built using the `amber_extfilt.a` module in *<Xcode>*`/Examples/CHUD/Amber/ExternalTraceFilter/`. Differences between TT6 and TT6E format as well as the specifics of the FULL trace format are detailed in *Amber Trace Format Specification v1.1* (*<Xcode>*`/ADC Reference Library/CHUD/AmberTraceFormats.pdf`). See `amber` man page.
`simg4`	A cycle-accurate simulator of the Motorola 7400 processor that takes TT6 (not TT6E) traces as input. See `simg4` man page.
`simg5`	A cycle-accurate simulator of the IBM 970 processor that takes TT6 (not TT6E) traces as input. See `simg5` man page.

Documentation and Help Tools

Table C-14 lists applications and command-line tools for creating or working with documentation and online help. These tools are located in *<Xcode>*`/usr/bin` and `/usr/bin`.

Table C-14 Documentation and help tools

Tool	Description
`compileHelp`	Merges contextual help RTF snippets into one file. This tool is included to support legacy applications. New contextual help projects do not use this tool. See `compileHelp` man page.
`gatherheaderdoc`	Gathers HeaderDoc output, creating a single index page and cross-links between documents. See `gatherheaderdoc` man page.
`headerdoc2HTML`	Generates HTML documentation from structured commentary in C, C++, and Objective-C header files. The HeaderDoc tags and scripts are described at http://developer.apple.com/darwin/projects/headerdoc/. See `headerdoc2html` man page.
`install-info`	Inserts menu entries from an Info file into the top-level dir file in the GNU Texinfo documentation system. It's most often run as part of software installation or when constructing a dir file for all manuals on a system. See http://www.gnu.org/software/texinfo/manual/texinfo/ for more information on the GNU Texinfo system. See `install-info` man page.

Localization Tools

Table C-15 lists the applications and command-line tools for localizing your own applications. These tools are located in *<Xcode>*`/usr/bin` and `/usr/bin`.

Table C-15 Localization tools

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

Version Control Tools

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

Subversion

Table C-16 lists the command-line tools to use with the Subversion system.

Table C-16 Subversion tools

Tool	Description
`svn`	The Subversion command-line client tool. You use this tool for manipulating files in a Subversion archive. See `svn` man page.
`svnadmin`	Creates and manages Subversion repositories. See `svnadmin` man page.
`svndumpfilter`	Filters data dumped from the repository by a `svnadmin dump` command. See `svndumpfilter` man page.
`svnlook`	Examines repository revisions and transactions. See `svnlook` man page.
`svnserve`	Accesses a repository using the `svn` network protocol. See `svnserve` man page.
`svnversion`	Summarizes the revision mixture of a working copy. See `svnversion` man page.

RCS

Table C-17 lists the command-line tools to use with the RCS system.

Table C-17 RCS tools

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

CVS

Table C-18 lists the command-line tools to use with the Concurrent Versions System (CVS) source control system.

Table C-18 CVS tools

Tool	Description
`agvtool`	Speeds up common versioning operations for Xcode projects that use the Apple-generic versioning system. It automatically embeds version information in the products produced by the Xcode application and performs certain CVS operations such as submitting the project with a new version number. For more information see the `agvtool` man page.
`cvs`	The latest tool for managing information in the CVS repository. (Note, this tool does not support CVS wrappers.) See the `cvs` man page for details. See also, `ocvs` below.
`cvs-wrap`	Wraps a directory into a GZIP format tar file. This single file can be handled more easily by CVS than the original directory.
`cvs-unwrap`	Extracts directories from a GZIP format tar file created by `cvs-wrap`.

Comparing Files

Table C-19 lists the command-line tools for comparing files.

Table C-19 Comparison tools

Tool	Description
`diff`	Compares two files or the files in two directories. See `diff` man page.
`diff3`	Compares three files. See `diff3` man page.
`diffpp`	Annotates the output of `diff` so that it can be printed with GNU `enscript`. This enables `enscript` to highlight the modified portions of the file. See `diffpp` man page.
`diffstat`	Reads one or more files output by `diff` and displays a histogram of the insertions, deletions, and modifications per file. See `diffstat` man page.
`merge`	Compares two files modified from the same original file and then combines all the changes into a single file. The `merge` tool warns you if both modified files have changes in the same lines. See `merge` man page.
`opendiff`	Opens FileMerge from the command line and begins comparing the specified files. See `opendiff` man page.
`patch`	Takes the output of `diff` and applies it to one or more copies of the original, unchanged file to create patched versions of the file. See `patch` man page.
`sdiff`	Compares two files and displays the differences so you can decide how to resolve them interactively. It then writes the results out to a file. A command-line version of FileMerge. See `sdiff` man page.

Packaging Tools

Table C-20 lists the applications and command-line tools used for packaging applications. These tools are located in *<Xcode>*`/usr/bin` and `/usr/bin`.

Table C-20 Packaging tools

Tool	Description
`codesign`	Creates a digital code signature for an application or software package. See `codesign` man page.
`CpMac`	Copies a file or a directory, including subdirectories, preserving metadata and forks. See `CpMac` man page.
`GetFileInfo`	Gets the file attributes of files in an HFS+ directory. See `GetFileInfo` man page.
`install`	Copies files to a target file or directory. Unlike the `cp` or `mv` commands, the `install` command lets you specify the new copy's owner, group ID, file flags, and mode. See `install` man page.

Tool	Description
`install_name_tool`	Changes the dynamic shared library install names recorded in a Mach-O binary. See `install_name_tool` man page.
`lipo`	Can create a multiple-architecture ("fat") executable file from one or more input files, list the architectures in a fat file, create a single-architecture file from a fat file, or make a new fat file with a subset of the architectures in the original fat file. See `lipo` man page.
`MergePef`	Merges two or more PEF files into a single file. PEF format is used for Mac OS 9 code. See `MergePef` man page.
`mkbom`	Creates a bill of materials for a directory.
`MvMac`	Moves files, preserving metadata and forks.
`SetFile`	Sets the attributes of files in an HFS+ directory. See `SetFile` man page.
`SplitForks`	Removes the resource fork in a file or all the resource forks in the files in a specified directory and saves them alongside the original files as hidden files (a hidden file has the same name as the original file, except that it has a "dot-underscore" prefix; for example `._MyPhoto.jpg`.). See `SplitForks` man page.

Scripting Tools

The tools listed in the following sections are located in *<Xcode>*`/usr/bin` and `/usr/bin`.

Interpreters and Compilers

Table C-21 lists the command-line script interpreters and compilers.

Table C-21 Script interpreters and compilers

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

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

Script Language Converters

Table C-22 lists the available command-line script language converters.

Table C-22 Script language converters

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

Perl Tools

Table C-23 lists the available command-line Perl tools.

Table C-23 Perl tools

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

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

Parsers and Lexical Analyzers

Table C-24 lists the available command-line parsers and lexical analyzers.

Table C-24 Parsers and lexical analyzers

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

Documentation Tools

Table C-25 lists the available command-line scripting documentation tools.

Table C-25 Scripting documentation tools

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

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

Java Tools

The tools listed in the following sections are located in `/usr/bin`.

General

Table C-26 lists the command-line tools used for building, debugging, and running Java programs.

Table C-26 Java tools

Tool	Description
`java`	Starts the Java runtime environment and launches a Java application. See `java` man page.
`javac`	The standard Java compiler from Sun Microsystems. See `javac` man page.
`jdb`	The Java debugger. It provides inspection and debugging of a local or remote Java virtual machine. See `jdb` man page.

Java Utilities

Table C-27 lists some of the applications and command-line tools for working with Java.

Table C-27 Java utilities

Tool	Description
`idlj`	Reads an Object Management Group (OMG) Interface Definition Language (IDL) file and translates it, or maps it, to a Java interface. The `idlj` compiler also creates stub, skeleton, helper, holder, and other files as necessary. These Java files are generated from the IDL file according to the mapping specified in the OMG document *OMG IDL to Java Language Mapping Specification, formal, 99-07-53*. The `idlj` compiler is documented at http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/toJavaPortableUG.html. IDL files are used to allow objects from different languages to interact with a common Object Request Broker (ORB), allowing remote invocation between languages. See `idlj` man page.
`javadoc`	Parses the declarations and documentation comments in a set of Java source files and produces HTML pages describing the public and protected classes, inner classes, interfaces, constructors, methods, and fields. See `javadoc` man page.
`javah`	Generates C header and source files from Java classes. The generated header and source files are used by C programs to reference instance variables of a Java object so that you can call Java code from inside your Mac OS X native application. See `javah` man page.

Tool	Description
native2ascii	Converts characters that are not in Latin-1 or Unicode encoding to ASCII for use with javac and other Java tools. It also can do the reverse conversion of Latin-1 or Unicode to native-encoded characters. See native2ascii man page.
rmic	A compiler that generates stub and skeleton class files for remote objects from the names of compiled Java classes that contain remote object implementations. A remote object is one that implements the interface java.rmi.Remote. See rmic man page.
rmiregistry	Creates and starts a remote object registry. A remote object registry is a naming service that makes it possible for clients on the host to look up remote objects and invoke remote methods. See rmiregistry man page.

Java Archive (JAR) Files

Table C-28 lists the available JAR file applications and command-line tools.

Table C-28 JAR file tools

Tool	Description
extcheck	Checks a specified JAR file for title and version conflicts with any extensions installed in the Java Developer Kit software. See extcheck man page.
jar	Combines and compresses multiple files into a single Java archive (JAR) file so they can be downloaded by a Java agent (such as a browser) in a single HTTP transaction. See jar man page.
jarsigner	Lets you sign JAR files and verify the signatures and integrity of signed JAR files. See jarsigner man page.

Kernel Extension Tools

Table C-29 lists the command-line tools that are useful for kernel extension development. These tools are located in /usr/sbin and /sbin.

Table C-29 Kernel extension tools

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

I/O Kit Driver Tools

Table C-30 lists the applications and command-line tools for developing device drivers. These tools are located in *<Xcode>*`/usr/sbin`.

Table C-30 Driver tools

Tool	Description
`ioreg`	A command-line version of I/O Registry Explorer. The `ioreg` tool displays the tree in a Terminal window, allowing you to cut and paste sections of the tree.
`ioalloccount`	Displays a summary of memory allocated by I/O Kit allocators listed by type (instance, container, and `IOMalloc`). This tool is useful for tracking memory leaks.
`ioclasscount`	Shows the number of instances allocated for each specified class. This tool is also useful for tracking memory leaks.

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.

Accessibility The technology for ensuring that disabled users can use Mac OS X. Accessibility provides support for disabled users in the form of screen readers, speech recognition, text-to-speech converters, and mouse and keyboard alternatives.

ACLs Access Control Lists. A technology used to give more fine-grained access to file-system objects. Compare with permissions.

actions Building blocks used to build workflows in Automator.

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

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

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

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

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

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

AppleScript An Apple-defined scripting language. AppleScript uses a natural language syntax to send Apple events to applications, commanding them to perform specific actions.

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

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

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

Aqua A set of guidelines that define the appearance and behavior of Mac OS X applications. The Aqua guidelines bring a unique look to applications,

integrating color, depth, clarity, translucence, and motion to present a vibrant appearance. If you use Carbon, Cocoa, or X11 to create your application's interface, you get the Aqua appearance automatically

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dashboard A user technology for managing HTML-based programs called widgets (see permissions). Activating the Dashboard via the F12 key displays a layer above the Mac OS X desktop that contains the user's current set of widgets.

Dashcode A graphical application used to build and debug Dashboard widgets.

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

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

dispatch queue A Grand Central Dispatch (GCD) structure that you use to execute your application's tasks. GCD defines dispatch queues for executing tasks either serially or concurrently.

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

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

dyld See dynamic link editor.

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

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

dynamic shared library A library whose code can be shared by multiple, concurrently running programs. Programs share exactly one physical copy of the library code and do not require their own copies of that code. With dynamic shared libraries, a program

not only attempts to resolve all undefined symbols at runtime, but attempts to do so only when those symbols are referenced during program execution.

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

Ethernet A high-speed local area network technology.

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

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

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

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

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

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

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

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

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

Grand Central Dispatch (GCD) A technology for executing asynchronous tasks concurrently. GCD is available in Mac OS X v10.6 and later and is not available in iOS.

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

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

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

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

IDE An acronym meaning "integrated development environment". An IDE is a program that typically combines text editing, compiling, and debugging features in one package in order to assist developers with the creation of software.

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

information property list A property list that contains essential configuration information for bundles. A file named `Info.plist` (or a platform-specific variant of that filename) contains the information property list and is packaged inside the bundle.

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

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

Interface Builder A tool for creating user interfaces. You use this tool to build and configure your user interface using a set of standard components and save that data to a resource file that can be loaded into your program at runtime. For more information, see "Interface Builder" (page 133).

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

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

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

iSync A tool for synchronizing address book information.

Java A development environment for creating applications. Java was created by Sun Microsystems.

Java Native Interface (JNI) A technology for bridging C-based code with Java.

Java Virtual Machine (JVM) The runtime environment for executing Java code. This environment includes a just-in-time bytecode compiler and utility code.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Objective-C An object-oriented programming language based on standard C and a runtime system that implements the dynamic functions of the language. Objective-C's few extensions to the C language are mostly based on Smalltalk, one of the first object-oriented programming languages. Objective-C is available in the Cocoa application environment.

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

Open Computing Language (OpenCL) A standards-based technology for performing general-purpose computations on a computer's graphics processor. For more information, see *OpenCL Programming Guide for Mac OS X*.

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

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

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

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

PackageMaker A tool that builds an installable software package from the files you provide. For more information, see "PackageMaker" (page 142).

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

pasteboard Another name for the Clipboard.

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

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

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

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

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

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

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

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

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

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

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

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

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

pthreads The POSIX Threads package (BSD).

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

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

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

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

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

real time In reference to operating systems, a guarantee of a certain capability within a specified time constraint, thus permitting predictable, time-critical behavior. If the user defines or initiates an event and the event occurs instantaneously, the

computer is said to be operating in real time. Real-time support is especially important for multimedia applications.

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

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

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

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

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

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

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

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

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

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

SCM Repository Source Code Management Repositories. A code database used to enable the collaborative development of large projects by multiple engineers. SCM repositories are managed by specific tools (such as CVS and Subversion), which manage the repository and handle check-ins and check-outs of code resources by engineers.

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

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

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

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

Shark A tool for analyzing a running (or static) application that returns metrics to help you identify potential performance bottlenecks. For more information, see "Performance Tools" (page 148).

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

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

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

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

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

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

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

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

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

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

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

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

timer A kernel resource that triggers an event at a specified interval. The event can occur only once or can be recurring. Timers are one of the input sources for run loops. Timers are also implemented at higher levels of the system, such as CFTimer in Core Foundation and NSTimer in Cocoa.

transformation An alteration to a coordinate system that defines a new coordinate system. Standard transformations include rotation, scaling, and translation. A transformation is represented by a matrix.

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

UFS UNIX file system. An industry-standard file-system format used in UNIX-like operating systems such as BSD. UFS in Mac OS X is a derivative of 4.4BSD UFS. Its disk layout is not compatible with other BSD UFS implementations.

umbrella framework A system framework that includes and links with constituent subframeworks and other public frameworks. An umbrella framework "contains" the system software defining an application environment or a layer of system software. See also subframework.

Unicode A 16-bit character set that assigns unique character codes to characters in a wide range of languages. In contrast to ASCII, which defines 128 distinct characters typically represented in 8 bits, Unicode comprises 65536 distinct characters that represent the unique characters used in many languages.

vector graphics The creation of digital images through a sequence of commands or mathematical statements that place lines and shapes in a two-dimensional or three-dimensional space. One

advantage of vector graphics over bitmap graphics (or raster graphics) is that any element of the picture can be changed at any time because each element is stored as an independent object. Another advantage of vector graphics is that the resulting image file is typically smaller than a bitmap file containing the same image. Examples of vector-image file types are PDF, encapsulated PostScript (EPS), and SVG. See also raster graphics.

versioning With frameworks, schemes to implement backward and forward compatibility of frameworks. Versioning information is written into a framework's dynamic shared library and is also reflected in the internal structure of a framework. See also major version; minor version.

VFS Virtual File System. A set of standard internal file-system interfaces and utilities that facilitate support for additional file systems. VFS provides an infrastructure for file systems built into the kernel.

virtual address A memory address that is usable by software. Each task has its own range of virtual addresses, which begins at address zero. The Mach operating system makes the CPU hardware map these addresses onto physical memory only when necessary, using disk memory at other times. See also physical address.

virtual machine (VM) A simulated computer in that it runs on a host computer but behaves as if it were a separate computer. The Java virtual machine works as a self-contained operating environment to run Java applications and applets.

virtual memory The use of a disk partition or a file on disk to provide the facilities usually provided by RAM. The virtual-memory manager in Mac OS X provides either a 32-bit or 64-bit protected address space for each task (depending on the options used to build the task) and facilitates efficient sharing of that address space.

VoiceOver A spoken user interface technology for visually impaired users.

volume A storage device or a portion of a storage device that is formatted to contain folders and files of a particular file system. A hard disk, for example, may be divided into several volumes (also known as partitions).

volume format The structure of file and folder (directory) information on a hard disk, a partition of a hard disk, a CD-ROM, or some other volume mounted on a computer system. Volume formats can specify such things as multiple forks (HFS and HFS+), symbolic and hard links (UFS), case sensitivity of filenames, and maximum length of filenames. See also file system.

widget An HTML-based program that runs in the Dashboard layer of the system.

window server A systemwide process that is responsible for rudimentary screen displays, window compositing and management, event routing, and cursor management. It coordinates low-level windowing behavior and enforces a fundamental uniformity in what appears on the screen.

Xcode An integrated development environment (IDE) for creating Mac OS X software. Xcode incorporates compiler, debugger, linker, and text editing tools into a single package to streamline the development process. For more information, see "Xcode" (page 127).

Instruments An integrated performance analysis and debugging tool. Instruments lets you gather a configurable set of metrics while your application is running, providing you with visualization tools to analyze the data and see performance problems and potential coding errors within your software. For more information, see "Instruments" (page 135).

Document Revision History

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

Date	Notes
2009-08-14	Updated for Mac OS X v10.6.
2008-10-15	Removed outdated reference to jikes compiler. Marked the AppleShareClient framework as deprecated, which it was in Mac OS X v10.5.
2007-10-31	Updated for Mac OS X v10.5. The document was also reorganized.
2006-06-28	Associated in-use prefix information with the system frameworks. Clarified directories containing developer tools.
2005-10-04	Added references to "Universal Binary Programming Guidelines."
2005-08-11	Fixed minor typos. Updated environment variable inheritance information.
2005-07-07	Incorporated developer feedback.
	Added AppleScript to the list of application environments.
2005-06-04	Corrected the man page name for SQLite.
2005-04-29	Fixed broken links and incorporated user feedback.
	Incorporated porting and technology guidelines from "Apple Software Design Guidelines." Added information about new system technologies. Changed "Rendezvous" to "Bonjour."
	Added new software types to list of development opportunities.
	Added a command-line primer.
	Added a summary of the available development tools.
	Updated the list of system frameworks.
2004-05-27	First version of *Mac OS X Technology Overview*. Some of the information in this document previously appeared in *System Overview*.

Document Revision History

Index

Symbols

Numerals

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z