Porting to Mac OS X from Windows Win32 API

Cross Platform: Windows

2009-05-06

Contents

Tables

Porting to Mac OS X from Windows Win32 API

Apple provides many programming resources for creating Mac OS X applications, and which one you choose depends upon your needs, preferences, and constraints. If you are unfamiliar with the Mac OS X platform, the process of choosing the right approach can be time-consuming and confusing.

The purpose of this Guide is to get you started porting an existing procedural Win32 application written in C or C++ to Mac OS X. You are, of course, encouraged to explore Apple's various APIs and choose the ones that best fit your needs.

Your first step is to familiarize yourself with the Mac OS X hardware and software environments, which are very different from the Windows environment. There are two books that you absolutely must read: *Mac OS X Technology Overview* and *Apple Human Interface Guidelines*. A third book, Learning Carbon, is also highly recommended because it teaches Mac OS X development through the construction of a simple C-based Mac OS X application. You will find further details on these books in the "For Further Information" (page 15) section at the end of this introduction.

Organization of This Document

After a brief discussion of several important topics, this Guide will introduce you to the procedural APIs that you are most likely to use in the following areas:

- "Drawing 2D Graphics in Mac OS X" (page 17)
- "3D Graphics in Mac OS X" (page 31)
- "The Mac OS X User Interface" (page 67)
- "Using Text in Mac OS X" (page 61)
- "Internationalization" (page 35)
- "Printing for Carbon Applications" (page 43)
- "Networking in Mac OS X" (page 55)
- "Multiprocessing" (page 51)

Background

To best understand the recommendations presented in this guide for porting your Win32 application to Mac OS X, you first need to understand how Mac OS X hardware and software differ from their Windows counterparts. The resources listed in this document's introduction are your best source for this information. However, the sections below introduce key Mac OS X concepts that are important to know about before reading any of the other pages of this guide.

Mac OS X

Of course, the biggest news is Mac OS X itself and the fact that its foundation is Darwin, a BSD flavor of UNIX. This means that Mac OS X enjoys all the benefits of a robust UNIX system, including protected memory, preemptive multitasking, BSD system services, and the full BSD tool set.

Mac OS X implements many of the POSIX APIs, which gives you access to a number of powerful UNIX tools and APIs. In addition, you can call supported POSIX routines from the Carbon and Cocoa application environments (explained below) and, in some cases, vice versa.

Aqua and Finder

On top of its UNIX foundation, Mac OS X also includes the visual interface that users see and use every day. This includes the Aqua user interface and the Finder, the primary application through which users find and manage directories, applications, and documents. Macintosh users are very particular about how their applications look and behave, so you need to ensure that your ported application matches their expectations.

Although the Aqua interface looks very different from the Windows interface, the constituent elements of both platforms are very similar. The behaviors of some Aqua interface elements, however, differ subtly from their Windows components, so be sure to check *Inside Mac OS X: Aqua Human Interface Guidelines* to make sure that your application behaves the way that Mac users expect.

File System Architecture

Because of Mac OS X's UNIX foundations, file access is governed by the traditional UNIX owner/group/other permissions system; a user's ability to access and change files is governed by the user's access privileges and the permissions system.

In addition, Mac OS X enhances file system integrity and security through the use of multiple file-system *domains*. There are four domains on a computer running Mac OS X: user, local, system, and network. Each of these domains has separate needs regarding accessibility and security. Any location on the computer or network belongs to one of these four domains, and Mac OS X automatically sets the permissions on files based upon the domain that they're are a part of.

Although Mac OS X ships using Apple's Hierarchical File System Plus (HFS+), Mac OS X also supports the BSD standard file system format (UFS), NFS (an industry standard for networked file systems), ISO 9660 (used for CD-ROM), MS-DOS, smb (a Windows file sharing standard), AFP (Mac OS file sharing), and the UDF file system (used by DVDs).

One thing that should be pointed out concerns the case-sensitivity of the HFS+ file system. For historical reasons, HFS+ is case-insensitive but case-preserving (unlike most implementations of UNIX, which are always case-sensitive). Case-preserving means that the code that implements the HFS+ file system will never change the case of any of the letters in a filename; however, filename matching and comparisons are done in a case-insensitive way.

Application Environments

Mac OS X supports multiple application environments--that is, environments you can use to develop your applications. They are

- Cocoa--a set of object-oriented application APIs, implemented in Objective-C,
- Carbon--a set of procedural APIs evolved from the traditional Mac OS system interfaces,
- Java--the APIs of Sun Microsystems' Java language platform,
- BSD--the APIs available via the Darwin Open Source portion of Mac OS X,
- Classic--a runtime environment to enable users access to applications written for Mac OS 9 and earlier versions,

Wherever possible, it is recommended that you develop applications using the Cocoa environment. However, Carbon is a good choice when your application is already implemented in C or C++ code that has been written in a procedural (as opposed to object-oriented) style.

Carbon Event Handling

Event handling under Carbon should be familiar to anyone who knows how to program Win32 applications. Though the Carbon event model uses a different set of terms and has structural differences from its Win32 counterpart, the overall structure of a Carbon application is similar to that of a Win32 application. In both cases, the operating system sends to the application those events that belong to it and routes them to the appropriate targets. Each target has a software routine associated with it. When a target receives an event, its associated software routine either handles the event or hands it back to the operating system, which handles the event in a standard way.

Event References

The Carbon equivalent of a Win32 MSG structure is an *event*, which is manipulated through a data type called an *event reference*. In Carbon, an event is an opaque data structure (referenced using an `EventRef`), much like a Win32 window `HANDLE`--that is, you do not know how is represented internally, but the operating system provides ways to access data associated with it. Carbon enables you to get certain data from an `EventRef`:

- the event class, which specifies the source of the event (for example, the keyboard, a control, or the menu system)
- the event kind, which specifies a specific event within a given class of events
- the event time, which is the time at which the event occurred
- event-specific parameters, which vary depending on the event

Carbon provides routines for getting the event class, kind, and time. In addition, once you know which event you have, you can ask for the value associated with any of its parameters by using the Carbon Event Manager function `GetEventParameter`. You can use this function to get a parameter's value by specifying (among other things) the parameter's name and type. With `GetEventParameter`, you always know the type of the value returned.

Event Targets

Just as in Win32 applications, various interface objects in Mac OS X applications--for example, windows, controls, and drop-down lists--can receive events. These interface objects are called *event targets*. When an event occurs, its event reference is delivered to the most specific target, which may handle the event or pass it to a less specific target. For example, a mouse click delivered when the mouse pointer is over a window's button is delivered to the button, not the enclosing window (which is a less specific target).

Event Handlers

In Win32 applications, each window has a window procedure associated with its window class when the window is defined. When an event belonging to a window occurs, the system gives the event's data to the window procedure, and the procedure executes. If the procedure does not know how to handle the event, it causes the system to handle it by executing the `DefWindowProc` procedure.

The Win32 environment handles events using code that works at the window level--that is, a window procedure contains code that handles all the events belonging to that window class. In contrast, Carbon handles events at the event level--that is, you write code that handles an event or a group of events, then you attach it to a given event target.

Part of initializing a target involves installing all its event handlers, creating what is called a *handler stack*. When an event is sent to its target, the target passes the event to the handler stack, and the first handler that matches it gets to handle the event. Each event target has a default handler (placed at the bottom of the handler stack), which provides default behavior for a given class of target.

With its use of event handlers, Carbon provides a finer level of control than Win32 does. However, for ease of porting, you can adapt your existing window procedures to work well in the Carbon environment. You do this by installing it in such a way that it handles all the appropriate events for the given target.

The following table summarizes the structural differences between event handling on Win32 and on Mac OS X:

	Win32	**Mac OS X**
Mechanism for reporting events	MSG structure	event reference
Name of handler	window procedure	event handler
How handler is associated with target	specified as parameter when window class is defined	`InstallEventHandler` invoked during target (window, button, etc.) initialization
How specific event is identified	message field of `DefWindowProc`	event kind (derived from the event type)
How unhandled events are processed	window procedure calls `DefWindowProc`	event handler returns a "not handled" value

Interface Builder and XCode

An application bundle is a highly structured set of files that must be configured correctly for an application to work properly. It is recommended that you use Interface Builder and XCode for developing all your Mac OS X applications. These two programs include numerous features that

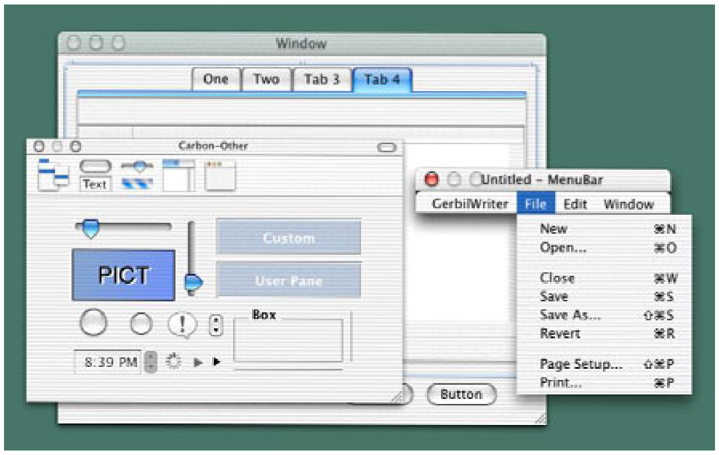
- streamline the process of designing, building, and debugging your application,
- organize the files needed by your application,
- enable you to create application bundles without knowing many of the low-level details involved,

Interface Builder

This application provides a GUI (graphical user interface) for the following actions:

- building your application's windows and menus from a palette of interface elements,
- customizing these elements to fit your needs,

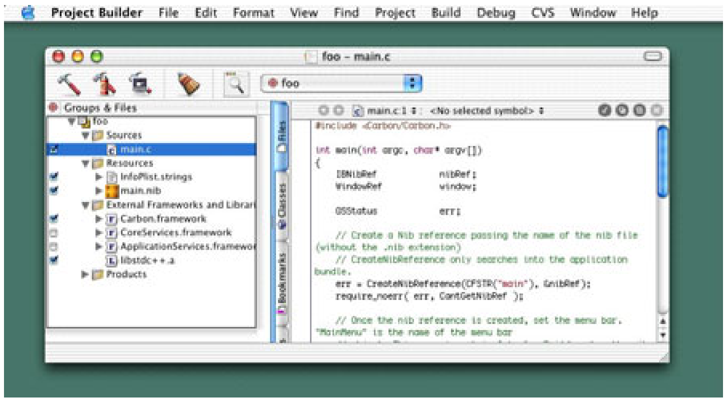
- giving these elements identifiers that your code will use to reference them,



You will use Interface Builder to translate your application's visual appearance into the visual appearance of a well-designed Mac OS X application. Interface Builder stores the results of your work in a "nib" file. Later, your finished application will read this file to re-create your application's visual appearance.

Xcode

Xcode is Apple's IDE (integrated development environment) for creating, designing, debugging, and deploying various kinds of Mac OS X software. For porting Win32 applications, you should use Xcode to create a nib-based Carbon application.



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

int main(int argc, char* argv[])
{
    IBNibRef            nibRef;
    WindowRef           window;

    OSStatus            err;

    // Create a Nib reference passing the name of the nib file
(without the .nib extension)
    // CreateNibReference only searches into the application
bundle.
    err = CreateNibReference(CFSTR("main"), &nibRef);
    require_noerr( err, CantGetNibRef );

    // Once the nib reference is created, set the menu bar.
"MainMenu" is the name of the menu bar
```

When you create a new project, Xcode creates a directory for your project and populates it with the appropriate folders and files. This directory contains all the files that are associated with your project, making it easy for you to backup your project or move it to another location.

Xcode uses the gcc compiler from the GNU Compiler Collection, so you can use Xcode for projects written in C, C++, and Objective-C (as well as Java and other languages). Xcode analyzes your source-code files at each build operation and creates the necessary make file automatically, so you don't have to maintain one yourself.

As part of the build operation, Xcode assembles the many files associated with a Mac OS X application into an application bundle. Once your project builds successfully, you can deploy it immediately by dragging the bundle's icon to the desired location.

Bundles

A bundle is a directory in the Mac OS X file system that stores executable code and software resources related to that code but is displayed in the Finder as a single icon. There are several kinds of bundles; you will be most interested in the *application bundle*, which is the kind of bundle used to package and distribute Mac OS X applications.

Bundles decrease user confusion and keep the file system from appearing cluttered. In addition, they help make sure that an application always uses the proper versions of any needed libraries, and they make it possible for a single application to contain support for multiple geographical locales.

When users double-click an application bundle's icon, Mac OS X executes the associated application. Mac OS X uses bundles to store an application and all the files it needs, including character strings in multiple languages, images, plug-ins, shared libraries, frameworks, and arbitrary data files (also called *resources*).

See Also

One of the primary purposes of this Guide is to direct you to the resources that will bring you up to speed on Mac OS X as quickly as possible. Unless you are very familiar with the architecture of Mac OS X, you should first read *Inside Mac OS X: System Overview*. A PDF of this book is freely available from Apple through the Internet.

Eventually, you will also need to read the book *Inside Mac OS X: Aqua Human Interface Guidelines* to understand the Mac OS X user interface. *Inside Carbon* makes a good second book to read because it introduces you to the actual process of creating a simple Mac OS X application.

This section, "For Further Information," ends each page of this Guide and offers pointers to the most relevant documentation and resources for porting your Win32 applications to Mac OS X. To make it as easy as possible for you to get started with Mac OS X, Apple makes all of its documentation and tools available free. (Apple charges for access to videotaped Worldwide Developer Conference sessions.)

Mac OS X Technology Overview	*Mac OS X Technology Overview* and
Inside Mac OS X: Aqua Human Interface Guidelines (REQUIRED)	*Apple Human Interface Guidelines*
Learning Carbon book	http://www.oreilly.com/catalog/learncarbon/ (published by O'Reilly)**(HIGHLY RECOMMENDED)**
Apple Developer Connection (Apple's support program for developers; includes free membership level and access to free Mac OS X Developer Tools suite)	http://developer.apple.com
Developer Tools page	http://developer.apple.com/tools/index.html
New to Carbon Programming page	*Carbon Overview*
Worldwide Developers Conference sessions on Mac OS X (available for purchase)	http://developer.apple.com/adctv/

Drawing 2D Graphics in Mac OS X

2D graphics are a big part of most applications. Except for the parts of a window drawn automatically by the user interface, you are responsible for drawing everything in your application's window. The portion of the Win32 API devoted to the Graphic Device Interface (abbreviated here as the Win32/GDI) provides routines that are analogous to those provided by QuickDraw (Apple's original graphic drawing environment for the Mac OS), but they are structured differently.

Since this Guide is written for programmers porting existing Win32/GDI code to Mac OS X, this document concentrates on explaining the QuickDraw API, the Mac OS X API that is the closest structurally to your existing code. However, depending on your situation, you may want to consider using the more powerful Quartz 2D API. See "Introduction to Quartz 2D," later in this document, for a brief description of Quartz 2D and a summary of its advantages, which include vector-based, resolution-independent graphics, sophisticated drawing based on paths and complex transformations, and built-in support for creating PDF documents.

A Caveat

This Guide is primarily concerned with the Win32/GDI API and Apple's QuickDraw API, both of which are well over 15 years old. During this time, these two APIs have evolved to provide graphics capabilities that have repeatedly tested the limits of the day's computers. The capabilities of both computers and their video displays have increased by more than a factor of 100 since these two drawing environments were created. This means that both of them are complicated by routines and concepts that, being outdated, clutter the API landscape and make it more difficult to describe how to move Win32/GDI drawing code to the Mac OS X platform.

Because of these limiting factors, this document necessarily omits some aspects of both drawing environments, especially those that are no longer in common use. For the sake of clarity, this document ignores certain special-case situations and simplifies certain details. For example, this document ignores the subject of indexed color, since virtually all computers now display 16 or 24-bit "direct" color. Also, this document refers to both device-dependent bitmaps (DDBs) and device-independent bitmaps (DIBs) as "bitmaps," and it does not mention how both platforms automatically map a "pure" color into the closest equivalent available on the display device being used.

The Mac OS X Graphics Architecture

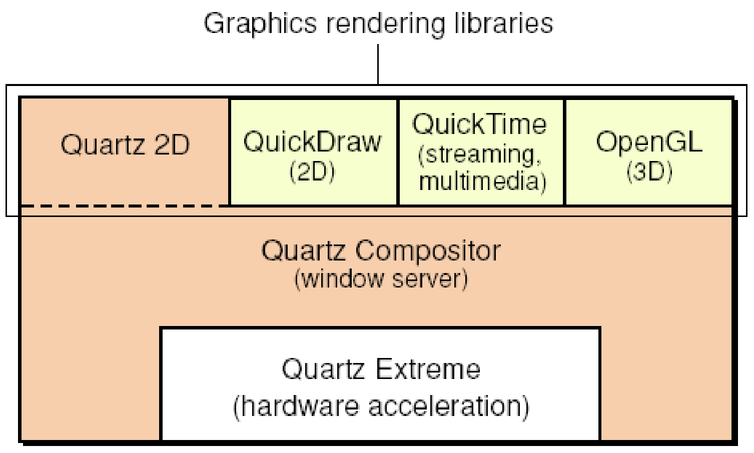
The core portion of the Mac OS X graphics and windowing environment is called Quartz. To achieve various benefits (including implementation of backward compatibility), Quartz is implemented in two pieces, a rendering API (Quartz 2D) and a window server (Quartz Compositor). This modularity enables Quartz 2D to coexist with other APIs, one of which is QuickDraw, the API that you are most likely to use when you port your Win32 application to Mac OS X.

Quartz 2D and QuickDraw

The core of the Mac OS X drawing and windowing architecture is called Quartz. It has two pieces:

- Quartz 2D--one of several graphics libraries that provide 2D text- and graphics-rendering services
- Quartz Compositor--the window server and Quartz Extreme, the component that interfaces to the computer's graphics-acceleration hardware

As you can see from the diagram below, QuickDraw is one of the graphics rendering libraries that you can use for drawing graphics under Mac OS X.


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

Under Mac OS X, QuickDraw thinks it is drawing directly to the graphics memory, just as it did in the earlier versions of Mac OS. In actuality, though, it is drawing to an off-screen graphics buffer. The Quartz Compositor combines this off-screen buffer with other graphics information (for example, soft-edge shadows and window transparency information) to produce the final pixmap. The Quartz Compositor delivers the final graphics image to the graphics hardware in a way that minimizes the delay in displaying changes to the screen, while maintaining the highest possible image quality (by preventing screen flicker and other visual artifacts).

Note : In older Mac OS X documentation, Quartz 2D is referred to as the Core Graphics Rendering component of Quartz, while the Quartz Compositor is referred to as the Core Graphics Services component of Quartz.

Color QuickDraw

Just as the Win32 platform added new graphics capabilities as display hardware became more powerful, so did the Mac OS platform. The original QuickDraw, which handled only one-bit black-and-white graphics, was replaced by the more powerful Color QuickDraw, which handles both indexed and direct color graphics. The Color QuickDraw API is an almost perfect superset of the original QuickDraw API, though some irregularities exist.

Inside Macintosh: Imaging with QuickDraw is the reference book you will use for learning QuickDraw. In it, the section on Color QuickDraw is an extension to the section on the original QuickDraw API; this means that you need to read both sections, even though you are interested in Color QuickDraw only.

Comparing Win32/GDI and QuickDraw

Two problems that confront anyone learning a new platform are learning the overall approach the new technology uses and relating concepts from the old technology to those of the new technology. This section addresses both concerns through the use of some introductory text about the most-used parts of the QuickDraw drawing environment and through tables that compare how the Win32/GDI and QuickDraw environment compare to each other.

Windows

While programming, you need a way to refer to a window. Under Win32, you identify a window through its *display context*. (Granted, in some cases you are manipulating the *device context* of the graphics hardware, but this Guide simplifies things by always using the term "display context.") In QuickDraw, you refer to a window by its `CGrafPort` (a programming shorthand for "color graphics port").

In QuickDraw, drawing commands have a slightly different form from their Win32 counterparts. While Win32 drawing commands always take a display context as their first argument, QuickDraw commands do not. Instead, QuickDraw always assume that any drawing command is intended for the current graphics port. Therefore, to draw into a different window, you must first "set the graphics port" (that is, set the `CGrafPort` of the desired window to be the current port), then issue the drawing commands for the new window.

There is one terminology difference you need to be aware of. Whereas the Win32/GDI API simply speaks of bitmaps, which can be black-and-white (one bit per pixel) or color (multiple bits per pixel), past and current versions of the Mac OS operating systems--and their documentation--use separate terms: *bitmap* for a one-bit black-and-white pixel image, and *pixmap* for a color pixel image.

Table 1 summarizes the differences regarding windows under the Win32/GDI and QuickDraw drawing environments.

Table 1 Comparison of window basics.

Win32/GDI	**QuickDraw**
The data structure that allows drawing in a window is a device context or a display context (abbreviated as DC).	The data structure that allows drawing in a window is a color graphics port, also called a `CGrafPort`.
A display context (usually) represents the client area of a window.	The `CGrafPort` represents the client area of a window.
All drawing routines include the name of the DC they are drawn to.	Drawing routines do not include the name of the window's `CGrafPort`. Instead, you make a window the current graphics port. All subsequent drawing occurs in the current port.
A memory device context is used as an offscreen bitmap.	To draw in an offscreen pixel map, you must create a "graphics world," or `GWorld` , and make it the current graphics port. Subsequent drawing occurs into the `GWorld` until the graphics port is changed.

It is possible, though not recommended, to access and change the bitmap associated with a DC.	It is possible to access and change the pixel map associated with a `CGrafPort`. See text for details.

If you are interested in accessing the pixel map associated with a `CGrafPort`, see the documentation for the following routines: `GetWindowPort`, `LockPortBits`, `GetPortPixMap`, `LockPixels`, `GetPixBaseAddr`, `GetPixRowBytes`, `UnlockPixels`, and `UnlockPortBits`.

Coordinate Systems

Apple has always controlled both the hardware and the operating-system software of its computers, so it was able to ensure that pixels are "square" – that is, that a pixel displayed on the screen is exactly as tall as it is wide. Because of this, the designers of QuickDraw did not have to deal with both logical and device coordinate systems, which possibly (in older computers) were not identical.

In QuickDraw, there is only one coordinate system, and it is a simple one. Unless you specify otherwise, the point (0,0) is the upper left corner of the client area of a window (if you are talking about the "local" coordinate system of a given window) or of the desktop itself.

Another subtle but very important difference between the Win32/GDI and QuickDraw coordinate systems is the relationship between the abstract coordinate grid and the pixels and lines drawn in relation to them. On the Win32 side, pixels (which have a measurable diameter) are drawn so that their centers correspond to the intersection of the appropriate horizontal and vertical grid lines, and the thickness of drawn lines straddle their corresponding coordinate lines.

In QuickDraw, pixels and lines are drawn in the spaces *between* coordinate lines. To be specific, they are drawn immediately to the right of and below the points that define them. The original QuickDraw engineers felt that this method leads to less confusion when questioning whether, for example, a 10-by-10 square drawn with its upper left corner at (0,0) includes the pixel corresponding to the point (10, 10). In QuickDraw, the answer is "yes," while in Win32/GDI, the answer is "no," and Win32 programmers must keep this "up to but not including" rule in mind.

Table 2 summarizes the differences regarding coordinate systems between the Win32/GDI and QuickDraw drawing environments.

Table 2 Comparison of coordinate systems.

Win32/GDI	**QuickDraw**
Device contexts can use logical coordinates that do not have a 1:1 relationship to the physical display (that is, scaling can occur).	Coordinates in the programming model have a one-to-one relationship to the physical display.
Has multiple mapping modes, in which increasing x values may indicate movement to the right or left, and increasing y values may indicate movement up or down.	Has fixed coordinate system: increasing x values move to the right, and increasing y values move down. Equivalent to Win32 MM_TEXT mode. Points may be described using either local (window) or global (desktop) coordinates.

Pixels are drawn at the intersection of the imaginary lines that define the logical coordinate system. That is, the imaginary lines that define point (x0, y0) go through the center of the pixel drawn at (x0, y0).	In QuickDraw, the intersection of the two imaginary lines that define point (x0, y0) is "ideal" — that is, it has no height or depth. Pixels, which have measurable height and depth, are drawn in the squares between the imaginary lines that define the coordinate system. In particular, the pixel corresponding to (x0, y0) is drawn to the right of and below the ideal point at (x0, y0).

Pens

Win32/GDI pens have evolved over time, so you now have different kinds of pens available for use — stock pens, logical pens, and geometric and cosmetic extended pens. In contrast, QuickDraw has only one kind of pen.

Just as QuickDraw always draws into the current port, it always draws with the current pen, and you use such routines as `PenSize`, `PenPixPat`, and `PenMode` to change the pen's size, pixel pattern, and drawing mode. If you have pens that you use frequently, you should create subroutines for defining them; that way, you can switch between pens with a single subroutine call.

In QuickDraw, a pen can draw in a solid color, optionally modified by an arbitrary one-bit pattern, or it can draw using an arbitrary pixmap. In addition, the drawing process can also be modified by the use of different transfer modes, though these are infrequently used.

The QuickDraw pen always has a pen pattern, a pen mode, a foreground color, and a background color, all of which affect how pen drawing works. The pen pattern, unlike the Win32/GDI hatch pattern, is defined by the programmer and can take on any value. QuickDraw defaults to using a solid black pen pattern and the `patCopy` pen mode, which results in an easy-to-understand pen that draws in the `CGrafPort`'s foreground color. However, the pen's behavior can become quite complex:

- If the pen pattern is a bitmap, depending on the pen mode used, a pixel "touched" by the pen can become the foreground color or the background color, it can be inverted, or it can remain unchanged.
- If the pen pattern is a pixmap, the pixmap itself is used to draw whatever line or area that is requested by the pen-drawing command. In this case, any pixel touched by the pen is replaced by the appropriate pixel from the pixmap.
- Also, when the pen is used to draw a line or outline a shape, it does not draw with its width straddling the path. Instead, the upper left corner of the pen is dragged along the path, meaning that "ink" is always deposited below and to the right of the current point on the path.

If your application makes heavy use of drawing features not found in QuickDraw, you should consider porting your Win32 application to Mac OS X using the Quartz 2D API. See "Using Quartz 2D," later in this document, for details.

Table 3 summarizes the most important differences regarding pens between the Win32/GDI and QuickDraw drawing environments. This summary should make the process of learning about QuickDraw pens easier.

Table 3 Comparison of Win32 geometric pens and QuickDraw pens.

Win32/GDI	QuickDraw

A DC includes attributes that point to the current pen and brush (which are separate data structures from the DC).	A `CGrafPort` contains attributes that describe the current pen and its color or pixel pattern.
The cross-section of a geometric pen is square.	The cross-section of a QuickDraw pen is rectangular--that is, its width and height can be different from each other.
A geometric pen can draw in a solid color and erase using background color.	A pen can draw in a solid color and erase using background color.
A geometric pen is specified by a width value; the line drawn is (usually) centered on the "ideal" line described by the drawing command.	A pen is described by width and height values; the pen draws in such a way that the upper left corner of the pen's rectangular cross-section traces the "ideal" line described by the drawing command.
A geometric pen, using a hatched brush, can draw a solid color modified by a predefined hatch pattern.	A pen can draw with a solid color and any pattern.
A geometric pen can specify dash, endcap, and line join styles.	QuickDraw has no equivalent, though Quartz 2D (a more advanced drawing environment) does.
By using a pattern brush, a geometric pen can draw using an arbitrary bitmap as its "ink."	By using a color pixel pattern (`PixPat`), a pen can draw using an arbitrary bitmap as its "ink."
The drawing process can be modified by using a DC's drawing mode attribute.	The drawing process can be modified by using different transfer modes.

Geometric Shapes

Though the Win32/GDI and QuickDraw environments have roughly the same capabilities when it comes to drawing geometric shapes, each environment structures the drawing process differently. QuickDraw does not have an equivalent to the Win32/GDI brush. Instead, when you are drawing shapes, you use a different command depending on how you wish to draw its interior, and the drawing can be accomplished with either the pen or an arbitrary bitmap or pixmap.

The verb at the beginning of each shape-drawing command's name indicates how the drawing is accomplished:

- Commands starting with "Frame" (for example, `FrameOval`) draw the outline of the shape using the pen and its behavior as described earlier; the interior of the shape is not affected.
- Commands starting with "Paint" (for example, `PaintOval`) paints the shape's interior using the pen and its behavior described earlier. The outline of the shape is not stroked by the pen.
- Commands starting with "FillC" (for example, `FillCOval`) fills the shape's interior using an arbitrary pixel pattern (not the pen pattern). Any pixel touched by the pen is replaced by the appropriate color pixel from the pixmap. The command for setting this *pixel* pattern is `PenPixPat`.

Table 4 summarizes the most important differences regarding geometric shapes between the Win32/GDI and QuickDraw drawing environments.

Table 4 Comparison of geometric shapes.

Win32/GDI	QuickDraw
Geometric shapes available: rectangles, ellipses, chords, pie-shaped wedges, rounded rectangles.	Geometric shapes available: rectangles, ovals, arcs and wedges, rounded rectangles.
To draw a basic geometric shape (for example, an ellipse) without filling it, you must draw it with the display context's brush filled to the background color. In this case, the drawing command is `Ellipse`.	To draw an outlined shape, QuickDraw uses commands that begin with "Frame" — in this case, `FrameOval`.
To draw an ellipse filled with a solid color, you must set the display context's brush to the given color and call the `Ellipse` command.	To draw a filled shape, QuickDraw uses commands that begin with "Paint" — in this case, `PaintOval`. This command paints the interior of the ellipse with the current foreground color. If you specify a one-bit pattern and a Boolean transfer mode, you can fill the oval with a result that is based on the pattern and weighted mixes of the foreground and background colors.
To draw an ellipse filled with a color bitmap, you must set the display context's brush to a pattern brush and call the `Ellipse` command.	QuickDraw uses commands that begin with "FillC" — in this case, `FillCOval`. This command requires, in addition to the description of the bounding rectangle, a reference to the pixel pattern to be used.
Win32 has an `InvertRect` command but no other commands for erasing or inverting geometric shapes.	QuickDraw has additional commands for erasing and inverting each geometric shape (in this example, `EraseOval` and `InvertOval`).

Polygons

In QuickDraw, the drawing of polygons mirrors the drawing of geometric shapes in most respects (see Table 5 (page 23), below). One difference from the Win32/GDI model is that in QuickDraw, you must explicitly create the polygon as a data structure before you can draw it.

Table 5 Comparison of polygons.

Win32/GDI	QuickDraw
A polygon is defined by an array of points, which is one of the parameters of the `Polygon` command.	In QuickDraw, you must create a polygon data structure before you can draw using it. You do this using an `OpenPoly` command, followed by line-drawing commands, followed by a `ClosePoly` command.
As with geometric shapes, you create outline, color-filled, and bitmap-filled polygons with the same command, but with different brush settings.	As with geometric shapes, you have multiple polygon commands. `FramePoly` draws the polygon with the pen. `PaintPoly` draws the polygon with the pan and fills its interior with a solid color. `FillCPoly` draws the polygon with the pen and fills its interior with a pixel map (color bitmap).

Win32/GDI has no commands for erasing or inverting a polygon.	As with geometric shapes, QuickDraw includes `ErasePoly` and `InvertPoly` commands.

Win32/GDI Paths, Win32/GDI Regions, and QuickDraw Regions

A QuickDraw region performs the same task as a Win32/GDI path in that you can use either to draw lines and fill areas based on a prerecorded set of graphic operations (see Table 6).

Table 6 Comparison of Win32/GDI paths and QuickDraw regions.

Win32/GDI	**QuickDraw**
A Win32/GDI path is the union of an arbitrary collection of graphical outlines that can be filled or stroked.	A QuickDraw region is a data structure that separates all points into those included in the region and those not included in the region. You can stroke its outline with a pen, paint its interior with a solid color or pixel pattern, use it as a mask for another drawing operation, or test for the presence of a mouse click inside it.
You must create a Win32/GDI path before you can draw using it. You do this using a `BeginPath` command, followed by a series of drawing commands, followed by an `EndPath` command.	You must create a QuickDraw region before you can draw using it. You do this using a `OpenRgn` command, followed by a series of drawing commands, followed by an `CloseRgn` command.
Use the `StrokePath` command to draw an outline using a Win32/GDI path and the current pen.	Use the `FrameRgn` command to draw an outline using a QuickDraw region and the current pen.
Use the `StrokeAndFillPath` command to draw with the current pen and fill with either a solid color or a bitmap.	Use the `PaintRgn` command to draw with the current pen and fill with a solid color. Use the `FillRgn` command to draw with the current pen and fill with a pixel map (color bitmap).
You can convert a Win32/GDI path to a clipping region for a given DC.	You can use a QuickDraw region as a clipping region for the current `CGrafPort` . You do not need to convert it to another form for this purpose.
There are no Win32/GDI equivalents to the commands mentioned at right.	QuickDraw includes `EraseRgn` and `InvertRgn` commands.

QuickDraw regions can also perform the same drawing and clipping tasks as Win32/GDI regions (see Table 7). When creating a QuickDraw region, you can use certain drawing commands that cannot be used to create a Win32/GDI region. However, this difference is largely negated by the fact that you can create a Win32/GDI path that is equivalent to the QuickDraw region, then convert it to a Win32/GDI region.

Table 7 Comparison of Win32/GDI regions and QuickDraw regions.

Win32/GDI	**QuickDraw**

A Win32/GDI region is a data structure that separates all points into those included in the path and those not included in the path. You can stroke its outline with a pen, paint its interior with a brush, use it as a mask for another drawing operation, or test for the presence of a mouse click inside it.	A QuickDraw region is a data structure that separates all points into those included in the region and those not included in the region. You can stroke its outline with a pen, paint its interior with a solid color or pixel pattern, use it as a mask for another drawing operation, or test for the presence of a mouse click inside it.
You create Win32/GDI regions like you do Win32/GDI paths, but your drawing operations are limited to ellipses, polygons, rectangles, and rounded rectangles. You can also convert a Win32/GDI path (which is more versatile) to a Win32/GDI region.	Like Win32/GDI regions, QuickDraw regions are created by a series of drawing operations, but QuickDraw regions can be created using all the shape-drawing commands.
You can create a region from the union, intersection, difference, or XOR of two other regions.	You can create a region from the union, intersection, difference, or XOR of two other regions.

Offscreen Bitmaps and GWorlds

On most computer systems, if you issue drawing commands that write directly to video memory, such commands will occasionally do so just as the video hardware is reading its memory and displaying the results. This results in unintended visual artifacts or "glitches" that degrade the quality of the display. The classic solution for this problem is to first draw the image in an offscreen bitmap, then transfer that image to video memory in one fast operation (called a bit-block transfer, or *bitblt*).

When Apple engineers created Color QuickDraw, they added a new data structure called a `GWorld` (a shortened version of the phrase "graphics world"). A `GWorld` behaves like a Win32/GDI offscreen bitmap, and in Color QuickDraw, you set the graphics port for both (on-screen) windows and (offscreen) and `GWorld`s with the command `SetGWorld`. You can learn about `GWorld`s and how to use them in the "Offscreen Graphics Worlds" chapter of *Inside Macintosh: Imaging with QuickDraw*.

As part of the Quartz drawing architecture, QuickDraw automatically draws into an offscreen bitmap; in fact, it cannot be used in any other way. Therefore, you do not need to use `GWorld`s if your only intent is to prevent visual artifacts from occurring in your display. However, there still are reasons to use `GWorld` s. If, for example, you need to draw a complicated image that you will be reusing, you can first draw the image into a `GWorld` and then save the result for reuse.

Bit-Block Transfers (BitBlts)

QuickDraw has one drawing command, `CopyBits`, that performs the same functions as the `BitBlt` and `StretchBlt` commands in Win32/GDI. Though `CopyBits` is most often used simply to copy a pixmap to new location, be aware that you can also use it to transform the image in various ways during the copy process; see the “Color QuickDraw” chapter of *Inside Macintosh: Imaging with QuickDraw* for details.

Table 8 summarizes the differences regarding bit-block transfers between the Win32/GDI and QuickDraw drawing environments.

Table 8 Comparison of bit-block transfer commands in Win32/GDI and QuickDraw.

Win32/GDI	**QuickDraw**

BitBlt transfers pixels from a source device context to a destination device context, modified by one of 256 raster operations that combines each source bit, the corresponding pattern bit (from the current brush), and the corresponding destination bit.	CopyBits transfers pixels from a source CGrafPort to a destination CGrafPort, modified by one of 18 source transfer modes that combines each source bit and its corresponding destination bit. If the source and destination areas are not the same size, CopyBits resizes the source area to fit the destination area.
StretchBlt behaves like BitBlt , but it stretches the source bitmap to fit the dimensions of the destination bitmap.	See above.
PatBlt copies a 1-bit pattern (from the current brush) into a destination device context, modified by one of 16 raster operations that combines each pattern bit with its corresponding destination bit.	You can use FillRect to achieve a similar result to that of PatBlt.
MaskBlt (not available on Windows 98 and earlier versions) transfers pixels from a source device context to a destination device context, modified by a 1-bit mask and a ROP4 raster operator	CopyMask and CopyDeepMask (available on all versions of Mac OS) transfer pixels from a source CGrafPort to a destination CGrafPort, modified by a pixel mask. Each resulting pixel is a blending of the source and destination pixels as determined by the value of the corresponding mask pixel. If the mask contains color pixels, the blending is computed per color component. CopyDeepMask behaves like CopyMask, but in addition, it allows you to select special transfer modes and to pass in a mask-clipping region (as in CopyBits).
PlgBlt (not available on Windows 98 and earlier versions) transfers pixels from a source device context to a destination device context, modified by a 1-bit mask and optional shearing, rotation, and mirror-image operations.	QuickDraw has no equivalent to PlgBlt, though Quartz 2D (a more advanced drawing environment) can perform arbitrary 2D transformations on pixmaps, including many not available to PlgBlt.

Metafiles and Picture Files

Just as the Win32/GDI drawing environment has metafiles, a mechanism for storing and playing back a set of drawing commands, QuickDraw has its equivalent: the *picture record* (also called a QuickDraw picture, or PICT). And, just as the original .wmf format was largely replaced by the enhanced .emf metafile, the original PICT format was largely replaced by a color-based "version 2" PICT format.

As with everything else in QuickDraw, PICT s are measured in pixels, and you can easily retrieve the size of the PICT from its header. You create PICT s by executing the OpenCPicture command, some drawing commands, then finally the CloseCPicture command. Later, you can draw them into a rectangular portion of a window or GWorld using the DrawPicture command.

The PICT format is also a standard format for the Clipboard. Along with the plain-text format, every Mac OS X application should be able to cut, copy, and paste PICT resources.

Text

Very little needs to be said about using QuickDraw to draw text, since you will probably use functions outside QuickDraw to do so. However, QuickDraw does include a `DrawText` command for simple text drawing. Just as there is only one pen associated with a `CGrafPort` , you also have only one "text tool" to work with, but you can change the font, style, size, and drawing mode of the current `CGrafPort` before using the `DrawText` command.

Table 9 summarizes the differences regarding text between the Win32/GDI and QuickDraw drawing environments.

Table 9 Comparison of basic text drawing.

Win32/GDI	**QuickDraw**
`TextOut` draws text in the current font to the location specified in the command's arguments.	`DrawText` draws text in the current font at the current pen location. (Use `MoveTo` to move the pen to a given location.)
`DrawText` draws text into a specified rectangle.	Use `TXNDrawCFStringTextBox` or `TXNDrawUnicode-TextBox` in the Multilingual Text Engine (MLTE) API to draw text into a specified rectangle.

An Introduction to Quartz 2D

Quartz 2D is Apple's newest two-dimensional drawing and windowing technology, available only under Mac OS X. It uses the Adobe PDF format as its internal graphics model, thus providing a rich 2D imaging model for graphics, as well as greatly simplifying the creation of PDF files.

Among the features available in Quartz 2D but not in QuickDraw are:

- vector-based, resolution-independent 2D drawing
- Bezier curves
- path-based drawing of graphics and text
- arbitrary 2D transformations of vector shapes (also called paths), text, and pixmaps
- transparency as an attribute of drawing
- contexts, an abstraction that enables the same drawing code to draw to the screen, a printer, or a PDF file
- per-context thread-safe drawing (all drawing for a given context must occur in the same thread)

Quartz 2D and Your Porting Effort

Porting an application to a new platform always has its own challenges, and these determine what development choices you have. When you port your Win32 application to Mac OS X, you have three choices:

- to port using QuickDraw only
- to port using QuickDraw, with occasional uses of Quartz 2D
- to port using Quartz 2D only

The Mac OS X implementation of QuickDraw includes two routines, `QDBeginCGContext` and `QDEndCGContext` , that enable you to switch to Quartz 2D temporarily for graphic operations that are not available in QuickDraw.

To compete with other Mac OS X applications similar to yours, you will eventually want to rewrite your application to use Quartz 2D exclusively. If your schedule allows it, you may want to consider porting your Win32 code directly to Quartz 2D instead of QuickDraw. This will give you a stronger initial product that will be easier to upgrade in the future.

Notes for Win32 Programmers

The following paragraphs describe several miscellaneous issues you should be aware of while porting a Win32 application to QuickDraw:

- Given the fact that QuickDraw graphics are always double-buffered under Mac OS X, the drawing process alone does not determine when the drawn graphics will become visible. If you want to control exactly when they are drawn to screen, first perform the drawing, then call the `QDFlushPortBuffer` routine.
- As mentioned earlier, you do not need to use `GWorld` s simply to prevent unsightly visual artifacts; Mac OS X's built-in double-buffering does this automatically.
- First-time users of the `CopyBits` routine may have trouble getting the desired result. If you're using `CopyBits` to copy a source pixmap, unchanged, to a destination pixmap, you need to make sure that the foreground color is set to black, the background color is set to white, and the source mode is `srcCopy` . For details, read the text at the top of page 4-34 of the *Imaging with QuickDraw* book.
- Because Apple controls the hardware on which Mac OS X runs, Mac OS X "knows" more about the display hardware available to it than the Win32 operating systems do. Mac OS X does not need to deal with device-dependent bitmaps (DDBs), device-independent bitmaps (DIBs), and DIB sections.
- Some Win32 programmers have reported that `CopyBits` , when used to stretch an image, produces results that are visibly different from those produced by `StretchBlt` . This difference occurs because of the different algorithms used to stretch an image. If you use `StretchBlt` in your Win32 code, be sure to check the visual appearance of the ported code that uses `CopyBits`.
- In indexed color modes under Mac OS X, black and white have fixed palette positions that cannot be changed. Unfortunately, the index values for black and white are reversed from what they are under Win32. If this situation applies to you, you will need to change your application so that it displays black and white correctly in its ported Mac OS X version.
- Although QuickDraw routines specify point coordinates with the horizontal component first--that is, with the x coordinate specified first--the QuickDraw points are stored in memory with the vertical component first--that is, with the y value stored at a lower memory location than the x value. Of course, you shouldn't be accessing memory directly anyway.

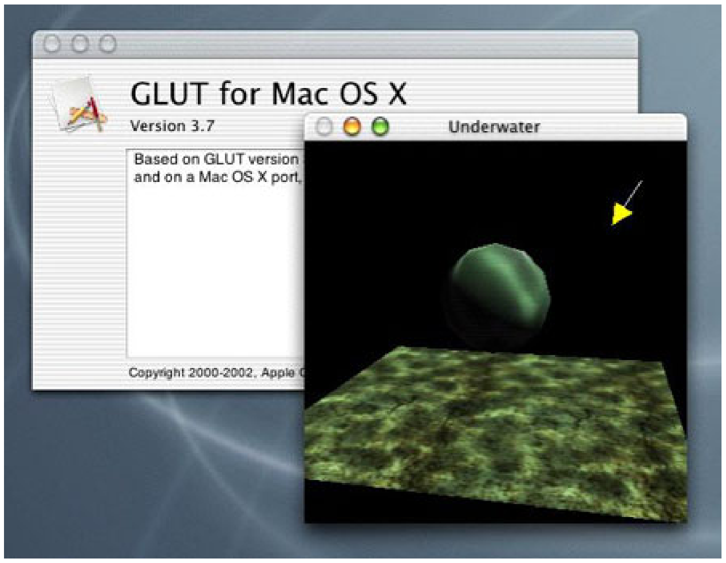
For Further Information

Your main reference for QuickDraw is *Inside Macintosh: Imaging with QuickDraw*. In addition to the link for this book, you may find some of the other links below helpful.

The QuickDraw documentation page (includes links to *Imaging with QuickDraw* and QuickDraw API documentation)	http://developer.apple.com/documentation/macos8/MultimediaGraphics/QuickDraw/quickdraw.html
QuickDraw Text book	*QuickDraw Text Reference*
links to `QDBeginCGContext` and `QDEndCGContext` documentation	*QuickDraw Reference*
QuickDraw Text Anti-Aliasing using Quartz 2D	http://developer.apple.com/qa/qa2001/qa1193.html
links to Quartz Primer, Drawing with Quartz 2D, Quartz 2D API reference	*Quartz 2D Reference Collection* *Quartz 2D Programming Guide* *Quartz Display Services Reference*
links to What's New with Quartz 2D documentation, Quartz Extreme information	http://developer.apple.com/quartz/

3D Graphics in Mac OS X

3D graphics are integral to many game, animation, and modeling products, and Mac OS X provides top-quality support for 3D in the form of the cross-platform OpenGL graphics environment. Depending on the graphics acceleration hardware installed, Mac OS X provides full support for OpenGL 3D v1.3 (OpenGL v1.2 for Mac OS X v10.1x and earlier). If your application supports OpenGL, you should have no problem porting your OpenGL code to Mac OS X. Maya, a high-end photo realistic 3D animation system, and Quake III, one of the most popular first-person shooter games available today, are two examples of cutting-edge OpenGL programs that have been ported to Apple's implementations of OpenGL.



Apple's Implementation

OpenGL itself is a hardware-independent API that provides no support for windowing tasks or obtaining user input. Apple's implementation provides four APIs for working with OpenGL:

- NSGL, for use with the object-oriented Cocoa application environment
- the AppleGL Library (AGL), for use with the procedural Carbon application environment
- CGL (the core OpenGL API), for use with full-screen graphics applications only
- the OpenGL Utility Toolkit (GLUT), for use with legacy GLUT code

Since you are porting your C or procedural C++ code to Mac OS X, you will probably want to investigate the AGL API. It is a higher-level API that enables you to do graphics rendering inside a window. AGL automatically loads the necessary libraries for the routines that your application uses, as well as enabling you to choose the best renderer for a given pixel format. If you wish, you can select specific renderers or specify criteria by which the renderer is chosen. AGL also handles the choosing of renderers when a graphics image spans multiple monitors.

Porting Notes

Be aware that OpenGL does not allow direct access to any of its frame buffers. Instead, you must use the appropriate OpenGL functions, such as `glReadPixels`, to read the frame buffer into system memory. Apple has optimized the routines that access and work with OpenGL, and these routines provide higher performance than most programmers could achieve even if they had direct access to the OpenGL frame buffers.

OpenGL is a cross-platform standard, but be aware that not all hardware renderers support all the OpenGL extensions. At run time, applications must check the OpenGL version or extensions string for the current renderer to determine what features the current renderer supports.

Mac OS X version 10.2 (Jaguar) supports 33 new OpenGL extensions and includes a number of other improvements. You may want to require customers to have Mac OS X version 10.2 or later to run your application.

Since you are making your application cross-platform, consider using QuickTime to simplify your Win32 and Mac OS X code bases. QuickTime includes functions that enable you to open files in dozens of graphics formats. You can simplify your code on both platforms by using QuickTime to open texture files.

For Further Information

OpenGL is an open graphic standard implemented on Windows, Mac OS, Linux, and other platforms. The best web site for documentation, links, and other resources is the OpenGL web site, at http://www.opengl.org. In addition, you should use the resources listed below to get started with OpenGL on Mac OS X.

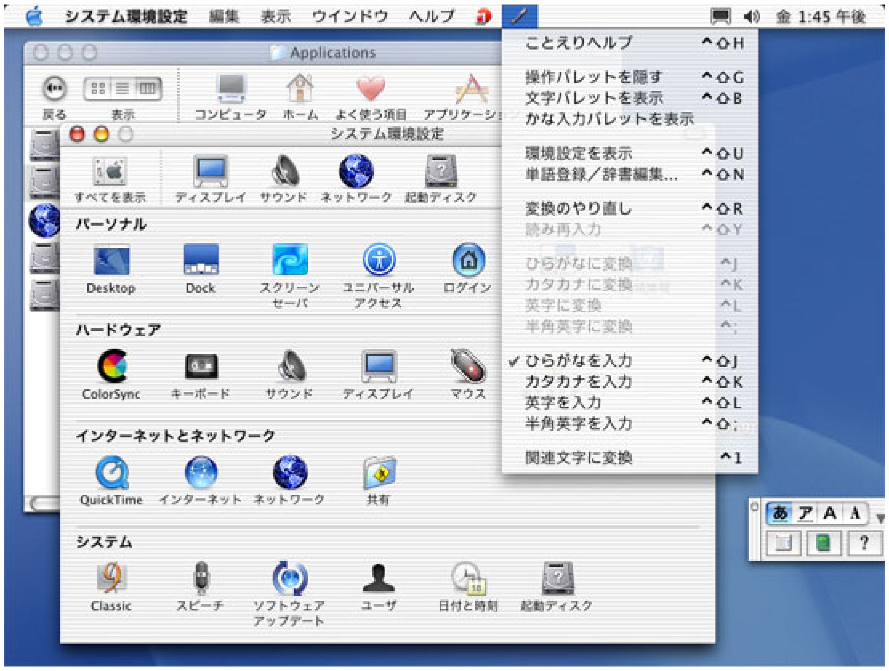
OpenGL for Mac OS book	*OpenGL Programming Guide for Mac OS X*
AGL API Reference	Inside Carbon: OpenGL
OpenGL Extensions Guide	http://developer.apple.com/opengl/extensions.html
list of OpenGL extensions supported by Mac OS X v10.3	http://developer.apple.com/opengl/panther.html
OpenGL sample code from the Mac OS X Development Tools suite	located on a Mac OS X hard disk at `/Developer/Examples/OpenGL/GLUT`
OpenGL Shader Builder and OpenGL Profiler tools	located on a Mac OS X hard disk at `/Developer/Applications`
OpenGL man pages	type "`man <commandname>`" from a Terminal window--for example, "`man glClear`" (see `gl.h` for command names)

OpenGL header files	located on a Mac OS X hard disk at `/System/Library/Frameworks/OpenGL.framework` and `/System/Library/Frameworks/AGL.framework`--in particular, `agl.h` (includes `gl.h`), `glu.h`, `glut.h`, `OpenGL.h` (for full-screen graphics), `glext.h` (for OpenGL extensions)
OpenGL sessions at WWDC 2002	* session 504--OpenGL: Graphics Programmability * session 505--OpenGL: Integrated Graphics 1 * session 506--OpenGL: Integrated Graphics 2 * session 513--OpenGL: Advanced 3D available for purchase at http://developer.apple.com/adctv/ * session 514--OpenGL: Performance and Optimization

Note: As on any UNIX system, you can access the OpenGL man pages by opening a Terminal window and typing `man commandname`. Header files are located in various places, so the best way to find one is to type `locate filename` in a Terminal window. If the command doesn't work, you need to build the underlying search database; see http://osxfaq.com/Tutorials/LearningCenter/UnixTutorials/WorkingWithUnix/page2.ws for instructions on how to do this.

Internationalization

Mac OS X is fully internationalized--that is, it has built-in support for the input, display, and manipulation of text in a wide variety of languages. Mac OS X also makes it easy for third-party applications to participate fully in its internationalized environment. Because of these factors, this Guide strongly recommends that even if your application does not initially support multiple languages or *locales* (culturally defined geographic regions), you should structure your application so that you can add such support later. Doing so takes little extra effort and promises considerable future rewards.



Background

Ever since the introduction of the Macintosh computer in 1984, Apple has committed itself to creating computers that support all of the world's major languages and locales (as well as many of the minor ones, too). Beginning with the Carbon application environment, which encouraged MacOS 9 developers to create their software in a way that would ease their transition to Mac OS X, Apple delivered a software architecture that provided even deeper support for localized software.

This section explains various terms and technologies related to Mac OS X internationalization. It is a necessary prelude to learning how to internationalize your Win32 application during the process of porting it to Mac OS X.

What Is Internationalization?

For the purposes of this discussion, we distinquish between the terms *localization* and *internationalization* as follows:

Localization is the process of changing an existing software product to make it suitable for use in a new language or locale. This process includes changing all user-visible text to a different language, replacing inappropriate images and sounds with locally-acceptable alternatives, and changing various formats and units of measurement to those used in a given locale. If the users of a given locale feel comfortable using an application because it meets their expectations regarding language, culture, and data formatting, you can say that the application has been successfully *localized* for that locale.

Internationalization (the subject of this page) is the process of designing and implementing a new software product so that it is easy to localize. This means taking advantage of operating-system features that facilitate localization instead of hard-coding the product to one locale.

Resources

Mac OS X resources, like their Windows counterparts, contain data needed by a software product--for example, text, icons, images, property lists, and data files. Unlike Win32 applications, which store resources in one or more dynamic link libraries (DLLs), Mac OS X stores some resources individually (for example, sound and image files), while other resources are stored in container files (for example, all of the application's text strings for a given language).

As you will see in the next subsection, resource files must be stored in specific locations. In addition, Mac OS X places restrictions on certain resources, including the specification of what resources must go into a given file and the name and location of that file.

Bundles

"Porting to Mac OS X from Windows Win32 API" (page 9) stated that a bundle is a directory in the file system that contains executable code and related resources. Mac OS X supports several varieties of bundles. The type of bundle of interest here is called an *application bundle*, which is a directory that contains all the executable code and resources needed to implement an application. To simplify matters for the user, the Finder treats an application bundle like an executable file, not a folder--when a user double-clicks it, the Finder runs the application (as opposed to opening the folder).

One key fact to keep in mind is that Apple's development tool suite assumes that your application bundle is structured according to certain guidelines. Following these guidelines simplifies your code and automatically makes your application internationalized--that is, it ensures that your application can easily be localized to additional languages and locales.

The following figure shows the directory structure of a typical application bundle.

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

The actual executable file is stored at `Contents/MacOS/MyApp`, while all the resources are stored in the directory `Content/Resources`.

In a bundled application, all the resources associated with a given language or locale are stored in a single directory, named <*language name or country or language abbreviation*>`.lproj`, within the Resources directory. For example, the directory for Japanese resources is named `Japanese.lproj` (`Japanese` is a language name), while the directory for the British version of resources for the English language is named `en_GB.lproj` (`en_GB` is a locale abbreviation, as specified by the ISO 3166 standard). A third way of specifying a localization directory involves indicating a language by its two-letter abbreviation, as specified by the ISO 639 standard; an example of this would be `en` for English.

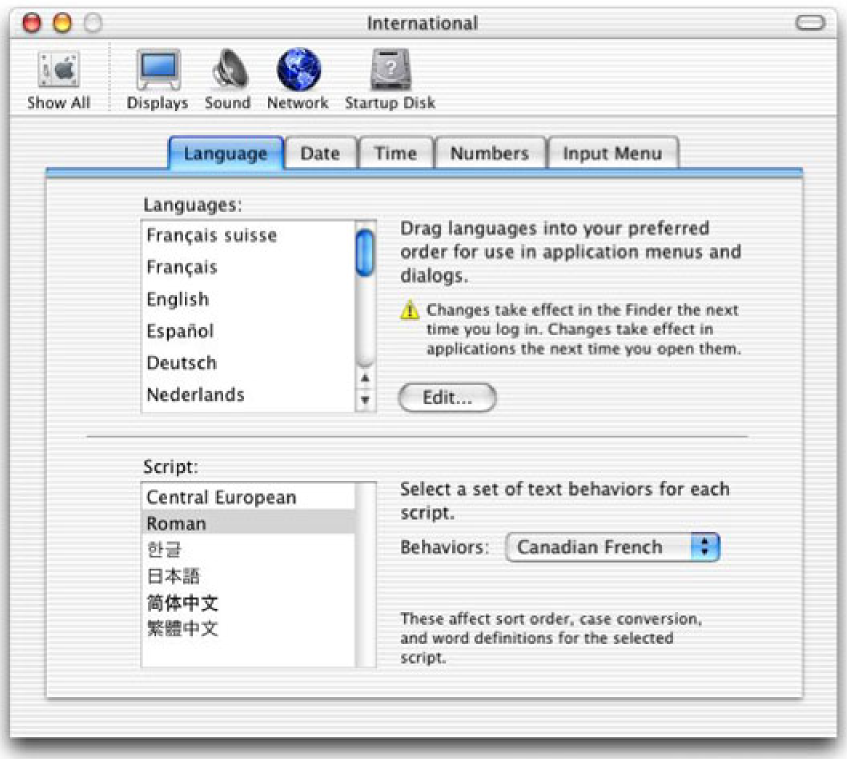
As expected, each file in a `.lproj` directory has a counterpart with the same name in every other `.lproj` directory. Each such file has its content specialized for a different language or locale, but it must have the same name as its counterparts.

The User and Developer Perspectives

Before you can understand how to add international support to your application, you must first understand how Mac OS X presents the issue of international support to both the user and the developer.

The User's View

Users express their language and locale preferences through the International module of the System Preferences application. In the Languages panel of this module (see below), users can drag and drop names in the Languages scrolling list to indicate which language they prefer to work in (the language at the top of the list) and, if that language is not available, what other languages they prefer to use.



Languages:
Français suisse
Français
English
Español
Deutsch
Nederlands

Drag languages into your preferred order for use in application menus and dialogs.

⚠ Changes take effect in the Finder the next time you log in. Changes take effect in applications the next time you open them.

Script:
Central European
Roman
한글
日本語
简体中文
繁體中文

Select a set of text behaviors for each script.

Behaviors: Canadian French

These affect sort order, case conversion, and word definitions for the selected script.

For example, with the settings shown above, the user has expressed her desire to see all software display text, numbers, and data according to the conventions assumed by people who identify with the Swiss dialect of French. If that is not possible, she prefers her software to display using French conventions. If that is not possible, she prefers English, Spanish, and so on down the list.

Mac OS X ensures that each application satisfies the user's language preferences to the best of the application's ability. One application, for example, may display its text and menus in Swiss French because it contains Swiss French resources. Another one (which lacks Swiss French, French, and Spanish resources but contains English resources) displays its text and menus in English because English is the closest that it can get to matching the user's preferences.

It is important to note this central assumption of Mac OS X users: All they have to do is set their language preferences in this one location. Once they have done this, *whenever users open an application, Mac OS X will automatically use the resources within the application that best match their language preferences*.

The Developer's View

Like the user, you also have a set of responsibilities that, once carried out, provide you with certain benefits. The benefit for you is the same as for the user: Mac OS X will automatically use the most appropriate resources to display your application's text and data as closely as possible to the user's language preferences.

To give a concrete example, you never have to write any code that checks some language-preference variable and, based on that value, retrieves the most appropriate version of the label for Button 3. Instead, you simply execute a procedure that tells Mac OS X to retrieve the label for Button 3. Mac OS X examines the user's language preferences and returns the Button 3 label string for the language that best matches the user's preferences.

Because of the way that Mac OS X handles language issues, Mac OS X applications are both "language-neutral" and "language-blind." Being language-neutral means that your application can display different languages equally well. Being language-blind means that it neither knows nor cares what language it is displaying to the user; it also means that your application contains no code that is tied to a given language.

Unicode

Though the topic of Unicode was covered in "Using Text in Mac OS X" (page 61), one piece of information bears repeating here: Unicode is the basis of all text storage and manipulation in Mac OS X. Unicode makes it possible for Mac OS X to use the same system and application code to display such widely different scripts as Arabic, traditional Chinese, Italian, and Russian.

Creating an Internationalized Application

This section talks about how to internationalize your application--that is, how to design and implement it to be language-neutral. (Localization, the process of modifying an application to make it acceptable to people in a given locale, is a complex process that will not be covered here.) This Guide recommends that you internationalize your application by following the steps below, even if your application initially supports only one language or locale.

You must do the following things to internationalize your application:

- Internationalize static user-interface resources.
- Internationalize dynamic user-interface resources.
- Write code that retrieves localized resources.

Internationalizing Static User-Interface Resources

Static user-interface resources are easy to find--they are the ones that you specify while building user-interface elements using Interface Builder. The output of Interface Builder is a file that ends with a suffix of `.nib`.

If a static user-interface element contains text or images that are locale-specific, the `.nib` file that contains them must be placed in the appropriate `.lproj` directory, and you must create a separate `.nib` file for each locale you are supporting. You will use Interface Builder to create each new version of the `.nib` while you are working on. For performance as well as for ease in localization, Apple recommends that you save all your menus in one `.nib` file and that you save each window or dialog in its own `.nib` file.

Languages vary by as much as 30 percent in terms of how much space a given phrase or sentence needs for it to be displayed on-screen. When you are localizing your user interface for additional languages, use Interface Builder if at all possible. By doing so, you can change an interface element's size and position as needed to accommodate any changes in text size.

Some resources, usually images and sounds, are the same for all locales but cannot be handled by using Interface Builder to insert them into your application. In such cases, you should store them directly in the Resources directory of your application bundle.

Internationalizing Dynamic User-Interface Resources

If user-visible text wasn't specified using Interface Builder, your code must somehow be displaying it dynamically. Mac OS X provides a way for you to write one set of code that will always retrieve the most appropriate localized string to your users. To do this, you must put all user-visible strings, formatted in a certain way, into a file with a name that ends with `.strings`. You must place this file within the appropriate `.lproj` directory, with one version of the file for each locale you are supporting. You can simplify your code slightly if you place these strings in a file named `Localized.strings`. A `.strings` file stores strings in key/value pairs, where the key is a string that is used to look up its associated value string, which is the actual string you want to display.

Most non-text resources have some meaning to the user (for example, the `house.jpg` image file in the application bundle shown earlier) and will be different for each culture. This means that they need to be placed inside the appropriate `.lproj` directory. This Guide recommends that, unless you absolutely know that a resource should be displayed the same way everywhere in the world, you place such non-text resources in `.lproj` directories.

Retrieving Localized Resources

To retrieve localized resources for use in your application, you will use functions from the CFBundle API. The structure of bundles is subject to change in the future. For this reason, it is important that you use functions from the CFBundle API for retrieving resources instead of hard-coding file pathnames into your application.

Retrieving Localized Strings

The CFBundle API uses the function `CFBundleCopyLocalizedString` to retrieve localized strings from a `.strings` file, although it is easier to use the supplied `CFCopyLocalizedString` macro, which calls `CFBundleCopyLocalizedString`. `CFLocalizedString` gets strings from the appropriate version of the current bundle's `Localizable.strings`, although variations of this macro can retrieve strings from other `.strings` files and other bundles.

When you hand `CFLocalizedString` the appropriate key string, it returns with the corresponding value string. For example, the Italian version of `Localizable.strings` for the Apple-supplied application TextEdit contains the following lines:

```
/* Message indicating file couldn't be opened; %@ is the filename. */
"Could not open file %@." = "Non posso aprire il documento %@.";
```

As a second example, the French version of Localizable`.strings` for TextEdit contains:

```
/* Message indicating file couldn't be opened; %@ is the filename. */
"Could not open file %@." = "Impossible d'ouvrir le fichier %@.";
```

When the TextEdit application wants to display a dialog to the user indicating that the requested file `foo.txt` cannot be opened, it calls `CFLocalizedString` with the key string `Could not open file %@.` as one of its arguments. If the user's preference is for Italian, `CFLocalizedString` returns the string `Non posso aprire il documento foo.txt.` If, however, the user's preference is for French, CFLocalizedString returns the string `Impossible d'ouvrir le fichier foo.txt.`

Retrieving Other Localized Resources

Because there are many types of non-string resources, the CFBundle API provides functions that return the location of the desired resource. Once your application knows the resource's location, it can then take the necessary steps to properly display the resource.

The functions that you will use most often are `CFBundleCopyResourceURL` and `CFBundleCopyResourceURLsOfType`. The first function returns the location of a specific resource. Using the example of the `MyApp` bundle, calling `CFBundleCopyResourceURL` and asking for the resource of type `jpg` named `house` causes the function to return the pathname to the correct localized version of `house.jpg`). The second function returns an array of locations for all localized resources of a given type (for example, all localized JPEG files). Again, using the MyApp bundle as an example, calling `CFBundleCopyResourceURLsOfType` and asking for all resources of type `jpg` causes the function to return an array of pathnames for the correct localized versions of `house.jpg` and `house-macos.jpg`.

Tools

Translating all the user-visible text of an application is one of the major tasks associated with localizing an application. Because language translators usually aren't programmers and because the process of manually changing text may introduce errors, tools exist to help the various parties involved in the translation process. Two such tools are AppleGlot, from Apple Computer, and PolyGlot, a third-party product.

AppleGlot can pull localizable information from Mac OS X software into a text file, enabling linguistic translators to easily translate the strings into the target language using only a text editor. AppleGlot can then reinsert the translated strings back into the application.

AppleGlot also supports incremental localization.When software already localized by AppleGlot is later enhanced or modified, subsequent use of AppleGlot highlights only the new or changed items that potentially require localization. AppleGlot supports Cocoa and Carbon applications, shared libraries, and frameworks.

Mac OS X expects to see all Unicode text encoded in the UTF-16 format. Some text editors store Unicode in the UTF-8 format, so be sure that anyone who handles user-visible text uses a text editor that can be configured to save text in the UTF-16 format. On the Mac OS side, one such text editor is Apple's TextEdit.

Notes for Win32 Programmers

The Windows platform stores Unicode text in "little-endian" form--that is, with the lower byte of the Unicode character in the lower memory location. The Mac OS X and UNIX platforms store Unicode in "big-endian" form, which stores the upper byte in the lower memory location. If you have Win32 (little-endian) Unicode text that you want to use when you port your application to Mac OS X, you must first convert the text to its big-endian equivalent.

Some Win32 applications contain their own custom support for multiple languages. If that is the case with your application, Apple recommends that you convert it to use localized `.nib` and `.strings` resources. However, if you do not have the resources to do that, you may want to port your existing international-support code to Mac OS X for the first version of your product, then add Mac OS X-style international support to your next release.

For Further Information

The links below point to the documentation you will need to get started with internationalizing your application.

Mac OS X Localization Page (including links to AppleGlot)	http://developer.apple.com/intl/localization/
Getting Started with Internationalization	*Getting Started with Internationalization*
Bundle Services Concepts and Tasks	http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles/index.html
Bundle Services API documentation	*Bundle Programming Guide*
PowerGlot	http://www.powerglot.com/

Printing for Carbon Applications

Printing under Mac OS X and Carbon is very similar to printing under Win32. In addition to having separate setup tasks before the printing begins and cleanup tasks after it has finished, the main print loop for both platforms is essentially the same:

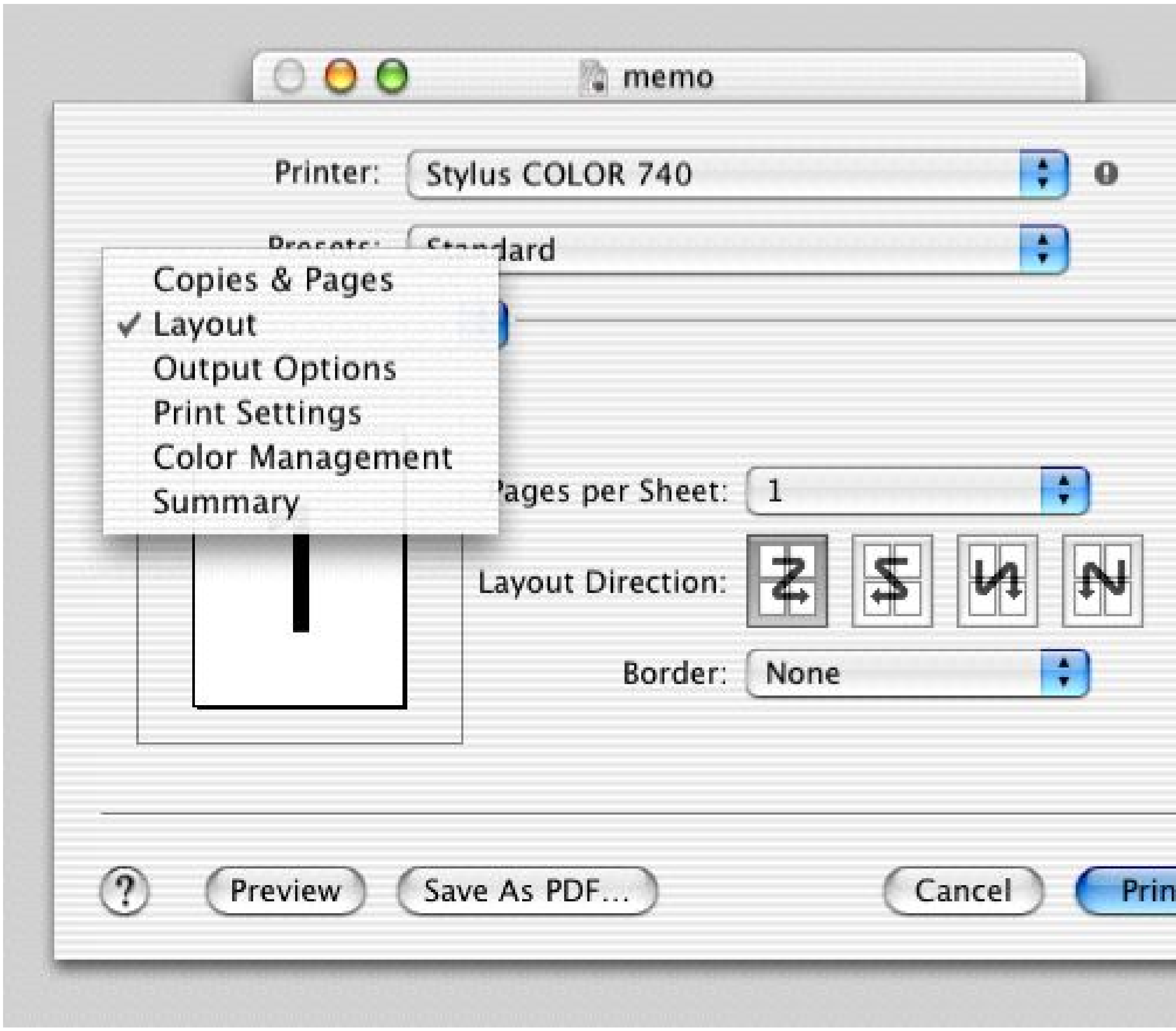
1. Declare the beginning of the print job (`StartDoc` for Win32, `PMSessionBeginDocument` for Mac OS X).
2. Declare the beginning of a page (`StartPage` for Win32, `PMSessionBeginPage` for Mac OS X).
3. Issue (largely) the same drawing commands as you would to the screen, "drawing" instead to an image area meant for a printer.
4. Declare the end of a page (`EndPage` for Win32, `PMSessionEndPage` for Mac OS X).
5. Declare the end of the print job (`EndDoc` for Win32, `PMSessionEndDocument` for Mac OS X).

In addition, you must create routines that handle all the errors that might occur during printing.

The Printing User Interface

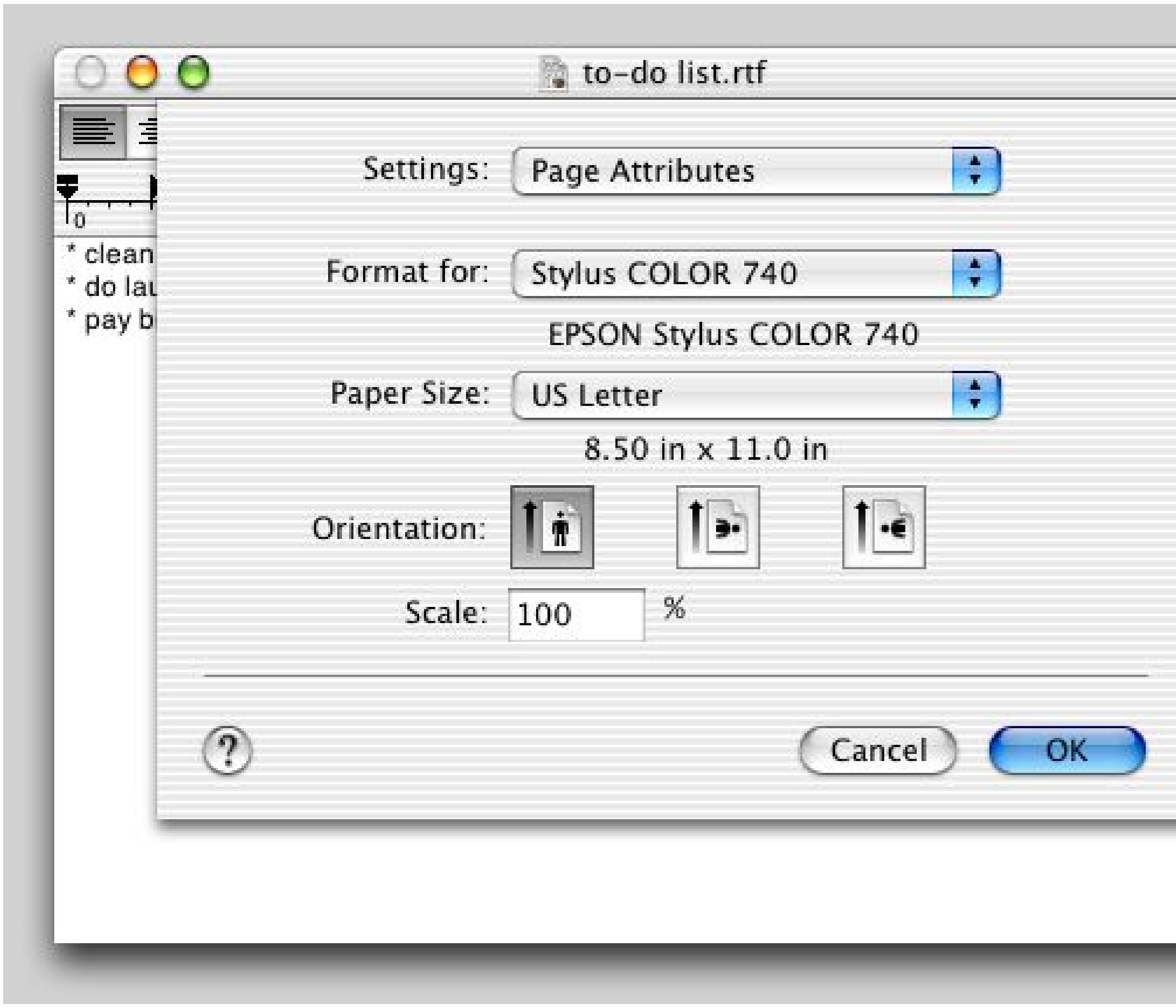
The three dialogs and windows that are associated with Mac OS X printing are not that different in function from their Windows counterparts. These are the Print dialog, the Page Setup dialog, and the Print Center windows.

The first, the Print dialog, enables the user to set the various parameters associated with the document about to be printed. This dialog can display different panes depending on which menu item user selects from the Features pop-up menu. Some panes are standard to all Print dialogs, while others are specific to the printer or the application. You can add your own panes by implementing them as printing dialog extensions (PDEs); see the *Extending Printing Dialogs* and *Printing Plug-In Interfaces Reference* books for details.



Users invoke the Page Setup dialog only if they want to change the default values of certain formatting properties. As with the Print dialog, you can add application-specific panes to the Page Setup dialog.

]



Settings: Page Attributes

Format for: Stylus COLOR 740

EPSON Stylus COLOR 740

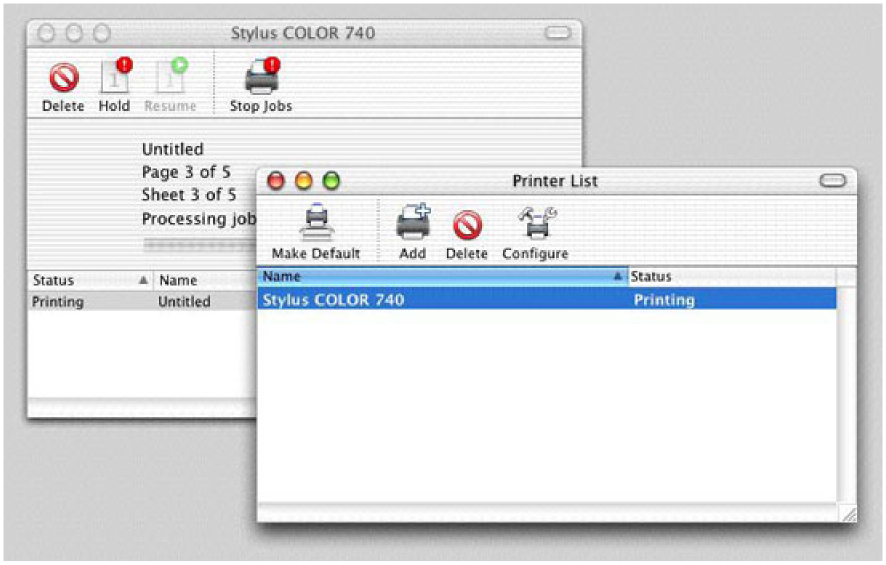
Paper Size: US Letter

8.50 in x 11.0 in

Orientation:

Scale: 100 %

Print Center is a utility supplied with Mac OS X; its function is to enable users to view and manipulate printers and their print jobs. This utility displays a single window named Printer List (see below). Users can see a printer's current print jobs and can suspend them, delete them, or change the order in which they print.



Carbon Printing Manager

The Carbon Printing Manager is the API you should use to implement printing when you port your Win32 application to Mac OS X, and the book *Supporting Printing in Your Carbon Application* is the best starting place for understanding how to do so. The subsections that follow give an overview of the actions your application must take to prepare for and perform printing. (Names in parentheses are the names of Carbon Printing Manager routines that you will use.) You will see that the process is similar in structure to that of printing in a procedural Win32 application.

Setting up the Page Format

When the user executes the Page Setup menu item, your application should do the following:

1. Create a printing session object (`PMCreateSession`).
2. Obtain a valid page format object for the document (custom routine from your application).
3. Specify that the Page Setup dialog should display itself as a sheet (`PMSessionUseSheets`).
4. Display the Page Setup dialog (`PMSessionPageSetupDialog`).
5. Save the values from this dialog for future use.
6. Release the printing session object (`PMRelease`) and handle any errors.

Executing the Print Command

When the user executes the Print menu item, your application should do the following:

1. Create a printing session object (`PMCreateSession`).
2. Create a a valid page format object (`PMCreatePageFormat`), if one doesn't exist.
3. Create a print settings object (`PMCreatePrintSettings).`
4. Display the Print dialog. (Here, the user clicks either the Print button or the Cancel button, and the appropriate actions occur.)
5. Release the printing session object (`PMRelease`), set the print settings object to `NULL`, and handle any errors.

When the user clicks the Print button, your application should execute its print-loop code.

The Print Loop

Your print-loop code should do the following:

1. Determine the maximum number of pages that can possibly be printed, then give that information to the computer.
2. Calculate the page numbers of the first and last pages to be printed using data from the Print dialog (`PMGetFirstPage`, `PMGetLastPage`), then give that information to the computer.
3. Create a new print job (`PMSessionBeginDocument`).
4. Set up a loop for drawing each page in the specified range. (Steps 5 through 12 are performed for each page.)
5. Tell the printing system that the code that follows begins a new page (`PMSessionBeginPage`).
6. Save the current graphics port (`GetPort`).
7. Get the graphics printing port for the page to be printed (`PMSessionGetGraphicsContext`).
8. Set that graphics port to be the current QuickDraw graphics port (`SetPort`).
9. Get the rectangle that defines the area in which drawing can occur (`GetPortBounds`).
10. Call the code that draws the current page.
11. Restore the previous graphics port (S`etPort`, using the value from step 6).
12. End the current page (`PMSessionEndPage`).
13. Once steps 5 through 12 have been performed for all the requested pages, signal the completion of the print job (`PMSessionEndDocument`).

Handling Errors

You need to write code that handles all the errors that might occur during printing. Part of this support includes a procedure to display alerts with their messages in the human language that most closely matches the user's language preferences. See "Internationalization" (page 35) for details on how to do this properly.

Save as PDF

Because Mac OS X stores document pages as PDF files during the spooling process, it is very easy to add support for saving a document as a PDF. You can do this as follows:

1. Direct your printing code to print to a file instead of a printer.
2. Add code that gets the desired filename and file location from the user.
3. Decide whether or not the Page Setup and Print dialogs should be part of the "save as PDF" process. If you decide they should not be (the usual case), you must add code that automatically adds reasonable page-related values to substitute for those normally provided by the Page Setup and Print dialogs.
4. Start the printing process.

By adding these changes, you can use the same code to print a document or save it as a PDF file.

For Further Information

The links below point to the documentation you will need to get started with implementing printing using the Carbon Printing Manager.

Mac OS X Printing page	http://developer.apple.com/printing/
Mac OS X Printing Documentation page	http://developer.apple.com/printing/
the Carbon Printing Manager documentation page	*Carbon Printing Manager Reference*
About the Mac OS X Printing System	*Mac OS X Printing System Overview*
Extending Printing Dialogs	*Extending Printing Dialogs*
Printing Plug-In Interfaces Reference	*Printing Plug-in Interfaces Reference*
Carbon Printing Manager header files	`PMApplication.h`, `PMCore.h`, `PMDefinitions.h` (see note below)
Printing sample code page (includes some samples for pre-Mac OS X versions)	Printing Sample Code

Note: Header files are located in various places, so the best way to find one is to type "locate <filename>" in a Terminal window. If the command doesn't work, you need to build the underlying search database; see http://osxfaq.com/Tutorials/LearningCenter/UnixTutorials/WorkingWithUnix/page2.ws for instructions on how to do this.

Multiprocessing

Dual-processor computers are now a significant part of Apple's hardware product line, and Mac OS X is designed to take advantage of them. If your Win32 application uses threading to take advantage of multiple processors, you can achieve similar performance when you port your application to Mac OS X. This section tells you how to get started.

For your information, Mac OS X supports multiple threading packages, including POSIX threads. (See Chapter 14 of *Inside Mac OS X: System Overview* for details.) If your Win32 application is already threaded, the Mac OS X API that most closely matches your code may be the Multiprocessing Services API, which is what this section covers.

Mac OS X/Win32 Similarities

Mac OS X and the Win32 platform take similar approaches to their support of multiprocessing, and the APIs involved are similar in structure and design. Here are the major similarities:

- Both approach the creation of multiprocessor-ready applications by splitting the application into multiple independent threads (called *tasks* in the Mac OS X Multiprocessing Services API), which the underlying operating system then schedules to run on multiple processors.
- Both implement symmetric multiprocessing (SMP).
- Both automatically assign threads/tasks to available processors in a way meant to maximize overall execution speed.
- On both platforms, applications created using threads/tasks run well on both single-processor and multiprocessor computers.
- Both use critical sections (called *critical regions* in Mac OS X) to restrict access to a given shared memory range to one thread/task at a time.
- Both provide semaphores for use as a synchronization mechanism among cooperating threads/tasks.

Approaches to Porting

There are three ways to port your Win32 threaded code to Mac OS X using the Multiprocessing Services API:

- writing "glue" code
- modifying existing code to use Multiprocessing Services routines
- rewriting your code to be more efficient

Which one you choose depends, of course, upon the priorities and limitations of your situation.

Before you decide on a porting approach, you should familiarize yourself with the Mac OS X APIs you will be using. As is the case on the Win32 platform, some APIs work in a multithreaded environment, others do not, and still others work if you observe certain restrictions. The issue of multithreading support may force you to change the approach you use to port your application to Mac OS X.

Writing Glue Code

In this approach, you leave your source code unchanged and concentrate on writing glue code that implements your multiprocessing APIs using Multiprocessing Services routines. This way, your code continues to run, believing that it is still operating on a multiple-processor Win32 computer.

The primary advantage of this approach is that if it is implemented correctly, you do not need to modify or retest the application code that uses threads, semaphores, critical sections, and so on. In addition, once you have the glue code working, you can port new Win32 applications to Mac OS X with minimal effort.

There are several significant disadvantages to this approach, however. First, the glue code necessarily introduces some processing overhead, and the ported application may run unacceptably slow. Second, writing the glue code is not a trivial task, and your schedule may not include the time needed to design, implement, and debug it.

Modifying Your Win32 Code

This approach involves leaving your program logic intact but replacing Win32-specific code with Mac OS X code that does the same thing.

The primary advantage of this approach over the glue-code approach is that the resulting code will run faster then it would using glue code. Depending on the complexity and length of your Win32 code, the porting process may be faster and easier.

This approach has two significant disadvantages. First, you will need to retest the ported application code. Second, this approach leaves you with two versions of your source code that must be maintained and enhanced separately.

Rewriting Your Win32 Code

On the Win32 side, the prevailing programming paradigm for multi-threaded applications centers around suspending threads that run too long and killing threads when necessary; these are actions that waste processor time needlessly. Applications that are structured around the producer/consumer model make better use of multithreading on any platform, and you should consider rewriting your Win32 code to use it.

An added advantage to switching to the producer/consumer model is that Mac OS X was designed to work well with it. When a task (thread) blocks, Mac OS X automatically suspends it quickly and with virtually no processor or memory overhead; when a task becomes unblocked, Mac OS X automatically and quickly reactivates it. Task creation and destruction, on the other hand, incur significantly greater overhead.

Tasks are suspended and resumed millions of times over their lifetimes, so these operations should be as efficient as possible. The producer/consumer model, in general, has no need to terminate a thread implementing a producer or consumer prematurely, though it switches tasks often. These two facts, taken together, explain why Mac OS X and the producer/consumer model are such a good fit.

The advantage of converting your program logic to use the producer/consumer model is that your application will run faster and will be easier to maintain on both the Win32 and Mac OS X platforms. The disadvantage is, of course, the time needed to rewrite and debug a major portion of your code.

The Multiprocessing Services API

A Multiprocessing Services task is a preemptively scheduled thread that is layered on top of a POSIX thread. The Multiprocessing Services API includes support for the following:

- *Tasks*: creating, terminating, and setting the relative priority of preemptive tasks
- *Critical regions*: creating, deleting, exiting, and attempting to enter critical regions
- *Semaphores*: creating, removing, signaling, and waiting on a semaphore
- *Memory allocation*: creating and manipulating a nonrelocatable block of memory available only to the threads of the current application (process)
- *Message queues*: creating, deleting, and manipulating FIFO message queues (used, among other things, for implementing producer/consumer programs)
- *Timers*: creating, arming, canceling, and deleting a timer; blocking a task until a specified time
- *Per-task storage variables*: setting and retrieving the value associated with a given index number and the current task
- *Processor availability*: querying the host computer about the number of processors available and their availability
- *Event groups*: creating, removing, and working with event groups (see below)

Event groups help you build code that blocks a thread until one of multiple events occurs. This is an improvement on the Win32 routine WaitForMultipleObjects in that, with an event group, you know which event or events have occurred and can make use of that knowledge.

For Further Information

You can find the the Multiprocessing Services API in the Carbon section of Apple's Developer Documentation website. For your convenience, a link to its page is listed below.

Multiprocessing Services	*Multiprocessing Services Reference*
books on pthreads	*Programming with POSIX Threads*, by David R. Butenhof (Addison Wesley, 1997) *POSIX 4: Programming for the Real World*, by Bill O. Gallmeister and Mike Loukides (O'Reilly, 1994)

Networking in Mac OS X

In most cases, networking means communicating with remote locations over the Internet, and that's what this document is largely about. If that's not what you're interested in, check "Related Subjects" toward the end of this document.

The rest of this document will talk about the various Mac OS X APIs that enable you to connect to and communicate with remote locations on the Internet:

- the BSD Sockets API, which gives you full access to UNIX-style sockets, for low-level access to networking functions.
- `CFSocket`, an Apple API that uses BSD sockets, but enables the handling of multiple socket connections within one thread.
- `CFNetwork`, an Apple API that simplifies the process of transmitting and receiving HTTP messages.

Before reading about these APIs, you first need to understand a concept called *run loops*.

Threads vs. Run Loops

Many socket-based programs are designed around the use of threads. For these programs, Apple provides similar functionality to facilitate porting the code to Mac OS X. Code using BSD sockets should require only minor changes to compile and run under Mac OS X.

One of the complexities of socket use involves the use of separate threads to handle the exchange of data. In a server environment, where hundreds of sockets are open simultaneously, the code that checks all these threads for activity may cause a significant performance hit. In addition, with threads, data synchronization is often an issue, and locks are often required to guarantee that only one thread has access to a given global variable at a time.

For such situations, Apple provides *run loops*, which behave like the message loop in a Win32 application. You can use run loops to handle various input sources, including sockets. Each thread of execution has exactly one run loop, which can handle multiple input sources. When you add an input source (for example, a socket) to a thread's run loop, you must also hand it a callback function that is to be called whenever the input source needs attention.

Event-driven code is more complex than code using blocking threads, but it delivers the highest network performance. When you use a run loop to handle multiple sockets in the same thread, you avoid the data synchronization problems associated with a multiple-thread solution. The `CFSocket` API contains routines that hand off the servicing of a socket to the thread's run loop.

Mac OS X Networking APIs

The following sections describe the three networking APIs you have available to you. The two higher-level APIs, `CFNetwork` and `CFSocket`, are both built on top of the third API, BSD Sockets. Each API is most useful in certain situations, so you need to decide which one to use. Documentation for these APIs resides in several places; see "For Further Information" at the end of this document for details.

BSD Sockets

BSD Sockets is the traditional UNIX API for connecting to other system processes and resources that reside on the Internet. If your Win32 code uses the Winsock API (which is itself based on BSD Sockets) for synchronous communication, you should have little trouble porting it to the Mac OS X implementation of BSD Sockets. The BSD Sockets API includes such procedures as `socket`, `bind`, `listen`, `accept`, `connect`, `read`, `write`, and `close`.

CFSocket

`CFSocket` provides a high-level API that uses BSD sockets to communicate with a remote Internet location, while also providing a configurable mechanism (using callback routines and the current thread's run loop) for automatically handling the data that the remote Internet location returns. For example you can:

- configure a `CFSocket` to automatically read incoming data or to call the callback routine when data is available to be read,
- configure a callback to be disabled or automatically reenabled after it has been triggered,
- manually enable or disable a callback,
- make the network connection in the background,

You will probably want to use the `CFSocket` API if your current networking code uses Winsock in its asynchronous mode.

CFNetwork

CFNetwork is a high-level networking API that makes it easy to create, send, and receive the kinds of messages that are commonly exchanged between clients and servers. It currently includes support for HTTP messages; support for other protocols (including SMTP, LDAP, and FTP) is planned for the future.

CFNetwork has functions for

- sending and receiving both HTTP requests and HTTP responses.
- getting and setting the various components of an HTTP message (for example, `CFHTTPMessageSetBody`, `CFHTTPMessageGetResponseStatusCode`).
- getting information about the message itself (for example, getting the message's HTTP version).
- creating HTTP streams for sending HTTP messages and receiving the response to them.

The following list gives you an idea of how to work with CFNetwork by showing you the routines you would use to create and send an HTTP request message.

- creating messages
  - create an HTTP request with `CFHTTPMessageCreateRequest`
  - set the message body with `CFHTTPMessageSetBody`
  - set the header with `CFHTTPMessageSetHeaderFieldValue`
  - verify the correctness of the headers by calling `CFHTTPMessageIsHeaderComplete`
  - serialize the message by calling `CFHTTPMessageCopySerializedMessage`
- sending messages
  - use `CFReadStreamCreateForHTTPRequest` to create an HTTP stream for transmitting an already created HTTP request message
  - Note: you can also send a message using a `CFSocket` or a BSD socket (discussed below)
- message cleanup
  - use `CFRelease` to dispose of the original message and its serialized version
- stream cleanup
  - close the stream using `CFReadStreamClose`
  - guarantee that the stream never uses its callback function by calling `CFReadStreamSetClient` with `NULL` arguments
  - prevent the stream from calling the existing callback using `CFReadStreamUnscheduleFromRunLoop`
  - release the reference on the stream using `CFRelease`

Related Subjects

Networking means different things to different people. The subsections below introduce two network-related resources you may find of interest. Pointers to relevant documentation are given in "For Further Information," below.

System Configuration Framework

This framework is used system-wide to allow various system resources to be configured under program control. In the context of networking, you might use this framework to configure network settings for the user and to change them automatically as network services are added and removed.

Network Services

Apple's Rendezvous technology, first made available in Mac OS X version 10.2 (Jaguar), enables both wireless and wire-connected computers, peripherals, and consumer devices to automatically connect to an ad hoc network. If you are interested in building this technology into your application, `CFNetwork` includes a Network Services API, `CFNetServices`, for doing so.

Network Services (also called `CFNetServices`) makes it possible for you to register your network service, along with a human-readable name and all the information needed to connect to your service. You can also create a network service browser, which enables you to browse for other registered services that you can connect to.

I/O Kit

If your interest in networking relates to writing software to communicate with associated networking hardware (for example, device drivers for a network interface card), you need to know about the I/O Kit. This is a collection of frameworks, libraries, tools, and other resources for creating device drivers for Mac OS X. The I/O Kit uses an object-oriented approach and is designed to make the creation of device drivers faster and easier.

For Further Information

Most of the documentation for Mac OS X's networking APIs is contained in header files, man pages, and books on UNIX network programming. See the notes below for help on accessing these forms of documentation.

CFRunLoop documentation in header file	CFRunLoop.h
BSD Sockets	See the man pages for individual commands. One recommended book on the subject is *UNIX Network Programming, Network APIs: Sockets and XTI, Volume 1, 2nd Edition*, by W. Richard Stevens, published by Prentice Hall, ISBN 0-13-490012-X.
Microsoft Winsock 2.2 API documentation (includes "Deviation from BSD Sockets" section)	ftp://ftp.microsoft.com/bussys/winsock/winsock2/WS-API22.DOC
CFSocket documentation in header file	`CFSocket.h`
CFNetwork documentation	*CFNetwork Programming Guide*
CFNetwork documentation in header files	`CFNetwork.h`, `CFHTTPMessage.h`, `CFHTTPStream.h`, `CFSocketStream.h`, and `CFNetServices.h`
Release Notes for CFNetwork and CFNetServices in Mac OS X v10.2	*CFNetwork Framework Release Notes*

Hardware & Drivers Documentation	Guides > Hardware & Drivers
I/O Kit Fundamentals book	*I/O Kit Fundamentals*
Networking sessions at WWDC 2002	* session 808--Managing I/O: CFRunLoop and CFStream * session 803--Mac OS X Networking Overview * session 805--Introducing CFNetwork available for purchase at http://developer.apple.com/adctv/

Note: As on any UNIX system, you can access the BSD Sockets commands by opening a Terminal window and typing "man <commandname>". Header files are located in various places, so the best way to find one is to type "locate <filename>" in a Terminal window. If the command doesn't work, you need to build the underlying search database; see http://osxfaq.com/Tutorials/LearningCenter/UnixTutorials/WorkingWithUnix/page2.ws for instructions on how to do this.

Using Text in Mac OS X

Given the Macintosh's strong roots in desktop publishing, it should come as no surprise that Mac OS X provides powerful APIs for manipulating and displaying multilingual, styled, static, and editable text. Of course, since you are porting an existing Win32 application to Mac OS X, you may not be interested in Mac OS X's advanced text features. Still, it's nice to know what's available, and that's what this section is about.

Depending on your situation, you may not even need know about the Mac OS X text APIs. If your use of text is limited to providing simple text-entry fields and labeling user-interface elements, you can do that from Interface Builder. If your application uses text in more sophisticated ways, you need to read this section.

Comparisons with Win32

The most important thing you need to know about Mac OS X's use of text is that, like the Win32 platform, Mac OS X always stores its strings in Unicode. Since your application probably uses Unicode for encoding text, you should be comfortable with Mac OS X's use of Unicode.

Like Win32, Mac OS X provides several APIs for static and editable text. If your needs are simple, Mac OS X provides easy-to-use routines that get the job done. Other, more complex APIs give you additional capabilities and greater control. Somewhere along this spectrum, you should able to find routines that roughly correspond to the Win32 routines in your original code.

Background

Although some Mac OS X routines take Unicode strings as arguments, most manipulate text stored as a `CFString` ("CF" is an abbreviation for Core Foundation, which is the part of Mac OS X that defines fundamental data structures and types used by Carbon and Cocoa). Here are some things to remember about `CFStrings`:

- A `CFString` is a "pure" string--that is, it does not include any text styles or formatting.
- A `CFString` "object" is an opaque data type that represents a string as an array of Unicode characters. Most text-manipulating APIs in Mac OS X use `CFStrings`.
- The String Services API (part of the Core Foundation software) includes the routines that manage converting between `CFStrings` and pure Unicode, comparing and searching `CFStrings`, manipulating `CFStrings`, and related functions.

Mac OS X provides built-in support for multilingual text, including

- multiple languages in the same document with no restrictions on language intermixing
- text highlighting that behaves correctly even in complex situations involving multiple languages of dissimilar type

- automatic support for Roman, Japanese, Semitic, Chinese, Indic, and other languages

Drawing Static Text

If you simply need to draw static text (that is, text that cannot be edited), your best choice is `DrawThemeTextBox`, which is part of the Appearance Manager API. This routine can only draw unstyled text, but the text is drawn in a font that matches a selected theme. Look for documentation on this and related routines in the file `appearance.h`. If the text you need to draw is part of your user interface, use the StaticText control on the Carbon-Controls pane of the Interface Builder controls palette.

Multilingual Text Engine (MLTE) contains more capable text-drawing routines. Its basic routine is `TXNDrawCFStringTextBox`, which takes a `CFString` as input. Since you are probably using Unicode strings in your application, you may want to use `TXNDrawUnicodeTextBox`, which takes a plain Unicode string as input.

For the most powerful text-drawing routines, you need to use Apple Type Services for Unicode Imaging (ATSUI). The text-drawing routine is `ATSUDrawText`, although you must first use other ATSUI routines to set up the drawing operation.

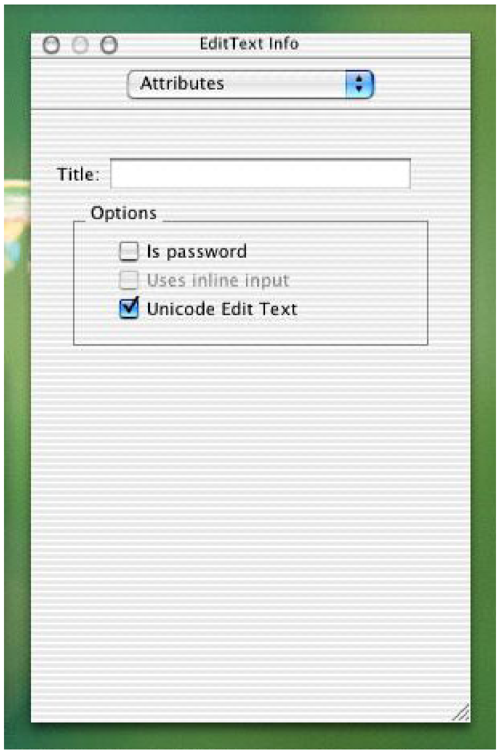
Drawing Editable Text

Mac OS X includes three APIs that you can use to provide editable text in your applications. The simplest, EditUnicodeTextControl, is sufficient for most situations. The second, MLTE, provides all the functionality most developers will need. Very few developers will need to use the third, ATSUI. The following sections describe these APIs.

EditUnicodeTextControl

The EditUnicodeTextControl is the simplest control for editable Unicode text. (Remember, Mac OS X uses Unicode strings throughout the system, so you need to be sure that you capture any text the user enters as Unicode text.) The routine you use to create an EditUnicodeTextControl is called `CreateEditUnicodeTextControl`; you can find it in the file `ControlDefinitions.h`.

If the text box you need is part of your user interface, use the EditText control on the Carbon-Controls pane of the Interface Builder controls palette. In the Attributes pane on the Info window for the EditText control, be sure to check the option titled "Unicode Edit Text."



Title:

Options

- [ ] Is password
- [ ] Uses inline input
- [✓] Unicode Edit Text

Multilingual Text Engine

MLTE provides a straightforward interface for providing multilingual Unicode styled text editing to your application. Built on top of ATSUI, MLTE includes support for

- document-wide tabs, justification, and margins
- text handling of any length (constrained only by memory)
- displaying text in all the languages that Mac OS X supports
- text-editing and drag-and-drop behavior as defined by the Apple user interface guidelines
- inline input of Japanese, Chinese, and other languages
- inclusion of sounds, images, and video within a text field
- scroll-bar handling and printing
- advanced font features
- 32 levels of undo and redo

Apple Type Services for Unicode Imaging (ATSUI)

ATSUI is the type technology that underlies all text drawing in Mac OS X. Although other APIs built on top of ATSUI provide simpler text-drawing functions, ATSUI provides an extremely high level of typographic control.

ATSUI enables sophisticated rendering of Unicode 3.2-encoded text (including such features as kerning, ligatures, and optical alignment). It automatically handles many of the complexities inherent in text layout, including how to correctly render text in bidirectional and vertical script systems.

ATSUI introduces new objects that enable it to provide higher levels of control and flexibility. Some examples are

- *style objects*, which are sets of font-related characteristics (called *style run attributes*) that are to be applied to a run of text
- *text layout objects*, which associate style runs, a buffer of text, and various default values to provide a software object that you can manipulate with ATSUI routines

ATSUI includes routines that enable you to perform various font, text styling, and layout functions. Among them are

- drawing and highlighting text based on the scripts (fonts/language conventions) involved
- discovering what fonts are available and which one most closely fits a set of requirements
- obtaining information about a given font and individual glyphs within a font
- creating and manipulating style objects and text layout objects
- manipulating style run attributes and associating them with runs of text
- determining the correct insertion point based on where the text has been clicked
- obtaining the spatial boundaries of a laid-out line of text
- calculating soft line breaks in a range of text

Because ATSUI uses Mac OS X's Quartz 2D graphics engine for all of its drawing, ATSUI users can access Quartz's image scaling, rotation, and transformation features.

One technical note: By reallocating and reusing `ATSUStyle` records, you can speed up the drawing of ATSUI-based text. See "Improving ATSUI text drawing performance" for details.

Other APIs

Mac OS X includes a number of other APIs related to text, fonts, and strings. In addition to the ones already discussed, here are the ones that you are most likely to use:

- Apple Text Services for Fonts (also called ATS): Used to determine what fonts are available on the current computer.
- Text Encoding Conversion Manager: Used to convert text between non-Unicode encodings and Unicode.
- Unicode Utilities: Used to determine text boundaries within a Unicode string, convert keystrokes to Unicode text, and determine the alphabetical order of two Unicode strings (taking into account the alphabetizing rules of the geographic locale under which the user is operating).

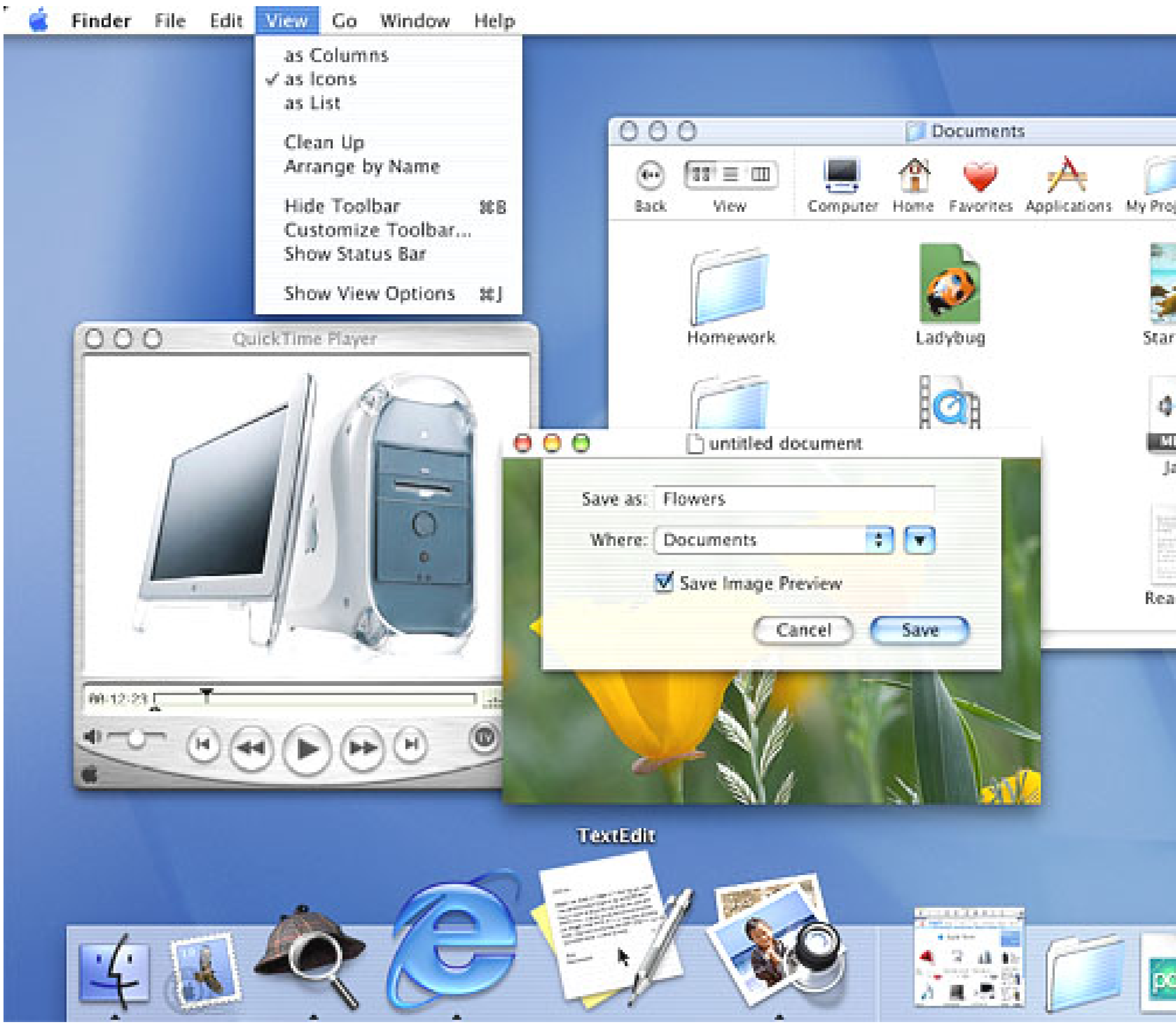
For Further Information

You can find all the APIs discussed here on the Carbon Developer Documentation page.

Appearance Manager	*Appearance Manager Reference*
Apple Text Services for Fonts (ATS)	*Apple Type Services for Fonts Reference*
Apple Text Services for Unicode Imaging (ATSUI)	*ATSUI Reference*
"Improving ATSUI text drawing performance" Technical Note	http://developer.apple.com/qa/qa2001/qa1027.html
Multilingual Text Engine (MLTE)	*Multilingual Text Engine Reference*
String Services (CFStrings)	*String Programming Guide for Core Foundation*
Text Encoding Conversion Manager	*Text Encoding Conversion Manager Reference*
appearance.h (DrawnThemeTextBox)	located on the main hard disk of a computer running Mac OS X, at `/System/Library/Frameworks/Carbon.framework/Frameworks/HIToolbox.framework/Headers/appearance.h`
ControlDefinitions.h	located on the main hard disk of a computer running Mac OS X, at `/System/Library/Frameworks/Carbon.framework/Frameworks/HIToolbox.framework/Headers/ControlDefinitions.h`
	Text and Fonts Sample Code
Unicode Utilities	*Unicode Utilities Reference*

The Mac OS X User Interface

The user interface is one of the most important aspects of your application. Whether you like it or not, the visual appearance of your application gives users a strong first impression of its quality and your programming skill. As users continue working with your application, its ease of use will, one way or the other, influence what people will say about not just your application but your company as well.



Save as: Flowers

Where: Documents

✓ Save Image Preview

Macintosh users are accustomed to software with a consistent, familiar, Mac-like interface. Does your application look like what your Mac customers expect it to look like? Do interface elements behave correctly, and are they in the right place? Will Mac users immediately feel comfortable with your application, or will they be confused and irritated by its nonstandard approach? One thing is certain: If you do a literal element-by-element porting of your Win32 application's user interface to Mac OS X, you will lose a lot of potential customers.

This document will give you the nuts-and-bolts information you need to know to start porting your application's user interface to Mac OS X. After that, the one book you need to read, cover to cover, is *Apple Human Interface Guidelines*.

But first, you need to learn more about your new customers.

Macintosh Users

Mac users are accustomed to user interfaces that are aesthetically pleasing, easy to understand, and easy to use. You should design your user interface with this in mind so your application will be appealing to a Macintosh User audience. Apple provides numerous resources discussing User Inteface Design that are presented on the Mac OS X User Experience web site at the address http://developer.apple.com/ue/. You should review the contents of this site and the guidelines presented there in detail when designing your appliction's user interface.

What can you do to make your ported MacOS X application a winner in the user interface arena? Get to know your new audience, and hire people who can help you. Here are some suggestions to get you started:

- Hire a consultant who has experience with Mac OS X user interface design. (Remember that Mac OS X is significantly different from previous versions of the Mac operating system, so recent experience is necessary.)
- Study the Mac OS X applications that will compete with yours.
- Study the applications that are a part of Mac OS X--for example, TextEdit and iTunes.
- Read a tutorial book on Mac OS X. (This is a good way to learn the basics that every Mac user takes for granted.)
- Read *MacAddict* and *Macworld* magazines, especially the software reviews.
- Visit a Macintosh users group and talk to people there.

Building a Mac OS X Application

Although it is possible to build a Mac OS X application using only a compiler and a text editor, it's definitely not a good use of your time. Apple provides a free, downloadable suite of powerful development tools for Mac OS X. The most important programs in this suite are Interface Builder, for creating the user interface, and Xcode, for writing, compiling, and debugging your application. (Other development environments are available for Mac OS X; CodeWarrior, from MetroWerks, is one such IDE.)

Since this document is concerned with the Mac OS X user interface, the overview of application development below emphasizes the use of Interface Builder, even though you will spend far more time with Xcode. This overview is necessarily brief.

Before you begin serious work with either Interface Builder or Xcode, you should study at least one of the following two books: *Learning Carbon*, available from the book publisher O'Reilly, or *Converter: Creating a User Interface with Interface Builder*, a short book available from the Apple web site. Each of these resources introduces you to the Mac OS X development process by walking you through the construction of a simple application.

Here are the steps you take to create a new Mac OS X application:

1. *Create a new Xcode project.* The Mac OS X development system organizes all the components related to a given application into something called a *project*. Begin by selecting the "New Project..." menu item, then choosing one of the project templates from the list presented. You will want to choose the "nib-based Carbon application" project template. (In a later step, you will use Interface Builder to build your user interface and then save it as a *nib file*. Later, your application will use the data from this file to create its user interface.)

2. *Open the nib file associated with this project.* You can do so by double-clicking on the file `main.nib` in the project window that appears after you have created the new project. This launches Interface Builder.

3. *Drag controls into your application window.* Interface Builder displays a default empty window and a prebuilt menu bar. You begin your design by dragging interface elements called *controls* from the control palette to the window. (An application can, of course, have multiple windows, but this description assumes a single-window application.)

4. *Set the attributes for each control in its Info window.* This process configures the control, setting its appearance and some of its behavior. In addition, you also specify information that later enables you to access the control from your application's code.

5. *Configure the menu bar and its menu items.* You can add and delete menus and menu items from the prebuilt menu bar. You then use the Info window to set the attributes for individual menu items.

6. *From Xcode, build and run the application.* The first time you build the application, Xcode generates the bundle that forms the basis of your application. Although the application has no content or custom behavior, it implements many default Mac OS X behaviors. Controls and menu items generate *events* (messages, in Windows terminology) when you interact with them.

7. *Write event handlers and other code.* You must write an event handler for each control and each application-specific menu item.

8. *Write code to implement the main window's event handler.* This enables controls to receive the events that belong to them.

9. *Debug your application.* Xcode includes a visual interface to gdb, the GNU debugger.

One important point: The best way to move your user interface to Mac OS X is to re-create it as part of the Xcode/Interface Builder development process. Fortunately, Interface Builder includes features that show you exactly where to place interface elements for them to conform to Apple's spacing and placement guidelines. Also, the Aqua interface includes two different sizes of certain interface elements (text fields and buttons, for example). Both these features make it easier and faster to design a good-looking window or dialog.

Differences Between Aqua and Win32

Aqua is the name of the distinctive Mac OS X user interface. Although its visual appearance is strictly 21st-century, it provides all the elements that characterize a modern graphical user interface. There are some elements in the Win32 user interface that do not appear in the Aqua user interface (as well as some in Aqua that are not available in Win32), but Aqua contains a streamlined set of elements that are more than adequate for any project.

It's important to know what the differences are between your current platform and Mac OS X. This section summarizes those differences, which will help you to be productive more quickly.

Terminology Differences

The first difference you need to know about is terminology. Certain elements are functionally identical (or almost so) on both the Win32 and Mac OS X platforms but are called by different names. Table 1, below, lists these terminology differences.

Table 1 Terminology differences between Win32 and Mac OS X UI elements.

Win32	**Mac OS X**
Desktop and Icons	
taskbar	Dock
Quick Launch bar	Dock
shortcut (icon)	alias
window button (in the Taskbar)	tile (in the Dock)
Windows and Dialogs	
Minimize, Maximize/Restore, and Close buttons	close, minimize, and zoom buttons
scroll box	scroller
size grip	resize control
dialog box	dialog
message box	alert
secondary window (a term that includes property sheets, inspectors, dialog boxes, palette windows, message boxes, and pop-up windows)	utility window (a floating, modeless window with no title bar; used for property sheets, inspectors, palettes, and other uses, but not dialogs)
Properties window (for folders and files)	Info window (for folders and files)
Open dialog box (used to browse the file system)	Open dialog
Save As dialog box	Save dialog

control panel	preference pane (a pane of the System Preferences application used to set the preferences related to a specific service)
Controls	
command button	push button
option button	radio button
check box	checkbox
list box	scrolling list
drop-down listbox	command pop-down menu
text box (edit control)	text input field
drop-down combo box	combo box
slider (trackbar control)	slider control
tree view control	data browser (list view)
"+/-" button in a tree view control	disclosure triangle
Miscellaneous	
ToolTip	help tag

One of the most pervasive terminology differences that you will encounter is the term *event*. In Mac OS X terminology, a Carbon event is the same thing as a Win32 message--namely, the data that a control receives when some system or user action occurs. (Because of its object-oriented nature, the Cocoa application environment handles events in a similar but not identical manner.)

UI Differences

As you can see from Table 2 and Table 3 (page 72) (below), both Win32 and Mac OS X have several user interface elements that the other does not. However, the differences are minor. You should be able to adapt your Win32 user interface to Mac OS X with little extra work. And who knows? You may find some of the Mac-only elements extremely useful in your ported application.

Table 2 Win32 UI elements with no Mac OS X equivalent.

Win32	**Notes**
Windows	
menu bar	The menu bar at the top of the Mac OS X Desktop performs the same function.
title-bar icon menu	The application's menu in the Mac OS X menu bar contains similar menu items.
Controls	

list view control	Can be created using the List Manager.
tree view control	The Mac OS X list-view data browser control is not an exact match, but you can use it for the same purposes as a tree view control.
rich-text box	Use the editable rich-text box in the Multilingual Text Editor (MLTE) API.
spin box	Consider using a text entry field or a combo box with a pop-up menu.
date-picker control	Can be created manually.
WebBrowser control	In most cases, the built-in Help Viewer can perform the same functions.
toolbar/status bar	Either create the toolbar within the window manually, or use utility windows.
Help	
balloon tip	Use help tag or alert, depending on the situation.

Table 3 Mac OS X UI elements with no Win32 equivalent.

Mac OS X	**Description**
Windows	
proxy icon	This is the icon immediately to the left of the window title. It represents the window's contents in drag-and-drop operations.
Controls	
column-view data browser	A control that enables you to navigate a hierarchical structure as a series of columns, with the items in one column representing the contents of the single highlighted item in the column to the left (see the second screenshot in the "Controls" section, below). The Mac OS X Finder uses this control as one way of viewing the contents of the hard disk.
pop-up menu	Similar to a drop-down combo box that allows only entries from the drop-down list box.
placard	Small informational panel, often placed to the left of a horizontal scroll bar; can include a pop-up menu.
bevel button	A large push button that can behave like a radio button or checkbox; it can also have a pop-up menu.
pop-up icon button	A variation on a pop-up bevel button.
image well	A rectangular area, displaying an icon or picture, that serves as a target for a drag-and-drop operation. The visual appearance of an image well is that the image is displayed in a rectangular depression slightly below the level of the window's surface.
disclosure button	Used in a window or dialog to display or hide additional information and controls that only some users will want to see.

relevance control	Used to indicate the relevance of a query result to the original query.
Miscellaneous	
Command key (on keyboard)	The key used for keyboard shortcuts; distinct from the Control key.

UI Discussion by Category

With the general discussion of Mac users, Aqua, and the Mac OS X development process completed, it's time to look more closely at the Mac OS X user interface and how it differs from the Win32 user interface. The subsections that follow, coupled with a thorough reading of *Inside Macintosh OS X: Aqua Human Interface Guidelines,* will give you the background you need to approach the Mac OS X user interface with confidence.

The topics that this section will cover are:

- the desktop
- the Dock
- icons
- menus
- windows
- dialogs
- controls
- mouse
- keyboard
- help
- miscellaneous

The Desktop

In its role as the idealized surface upon which windows, dialogs, and utility windows (palettes) lie, "the desktop" is familiar to both Mac OS and Windows users. As is the case with some Windows-based computers, certain Apple computers can be connected to multiple display monitors; in such cases, the desktop is said to "span" multiple monitors.

One of the most significant differences between Windows and the Mac OS (historically, as well as in Mac OS X) is the relationship on the desktop between a software application and the documents associated with it. From the beginning, the Apple user interface guidelines stressed the importance of using metaphors to everyday physical objects as a way of helping users understand unfamiliar software applications. As a result, most applications display each document in a separate window, which users find familiar because it mirrors their experience with paper documents. The emphasis is on the document, and the underlying software is largely invisible. The application itself does not have a window, and its presence is visible to the user only in the display's menus and menu items.

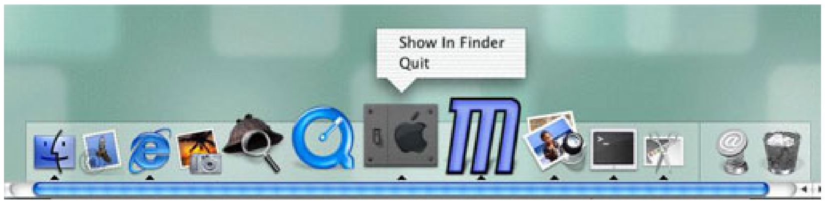
In contrast, Microsoft developed the Multiple-Document Interface (MDI) as the user interface for software applications that can manipulate multiple documents simultaneously. In this interface, the application has its own window, and its open documents may or may not be visible inside the client area of the application's window.

What does this mean for you as a developer porting your Win32 application to Mac OS X? Of course, this approach to UI issues makes Mac OS X applications easier to understand and use. From a programming standpoint, however, this different approach does not change your job much. Both platforms retain the same concepts: an application owning multiple, simultaneously open document windows, with input events being routed to the window that should receive them. Because of this, the basic architecture of your application does not need to change when you port it to Mac OS X.

If you have an MDI application, there is one structural simplification you will want to make. Since, on the Mac OS X side, there is no parent window enclosing your document windows, you will need to remove from your original code those menu items or commands that manage the visibility and placement of documents within the parent window (for example, menu items for tiling and cascading document windows).

The Dock

As a Win32 developer, you may not be familiar with the Mac OS X desktop object called the Dock, which is a "shelf" (usually at the bottom of the desktop) on which large application icons called *Dock tiles* are visible. The Dock always contains tiles for applications that are currently executing. However, it also contains tiles for applications that the user wants to keep easily accessible. The user can easily see which applications are currently executing, because they have a small black triangle next to their tile. Tiles are placed into the Dock by either the user or by Apple (although the Finder and Trash tiles are permanent occupants of the Dock).



As a developer, you have no influence over the Dock's contents. However, you can add menu items to the pop-up menu that appears when you press and hold down the left mouse button while pointing to your application's tile in the Dock.

For further information regarding the Dock, including important behaviors to implement, see the Mac OS X Environemnt chapter of *Apple Human Interface Guidelines*.

Icons

No visual element in Mac OS X is more distinctive than its photo-illustrative icons. Mac OS X icons are 128 by 128 pixels in size, with anti-aliasing, alpha channels, and translucency effects available to enhance the icon's appearance. Though they are, in fact, illustrations, they look like photographs.

Mac OS X helps users recognize the function of a given icon by specifying that each icon must belong to one of several icon genres:

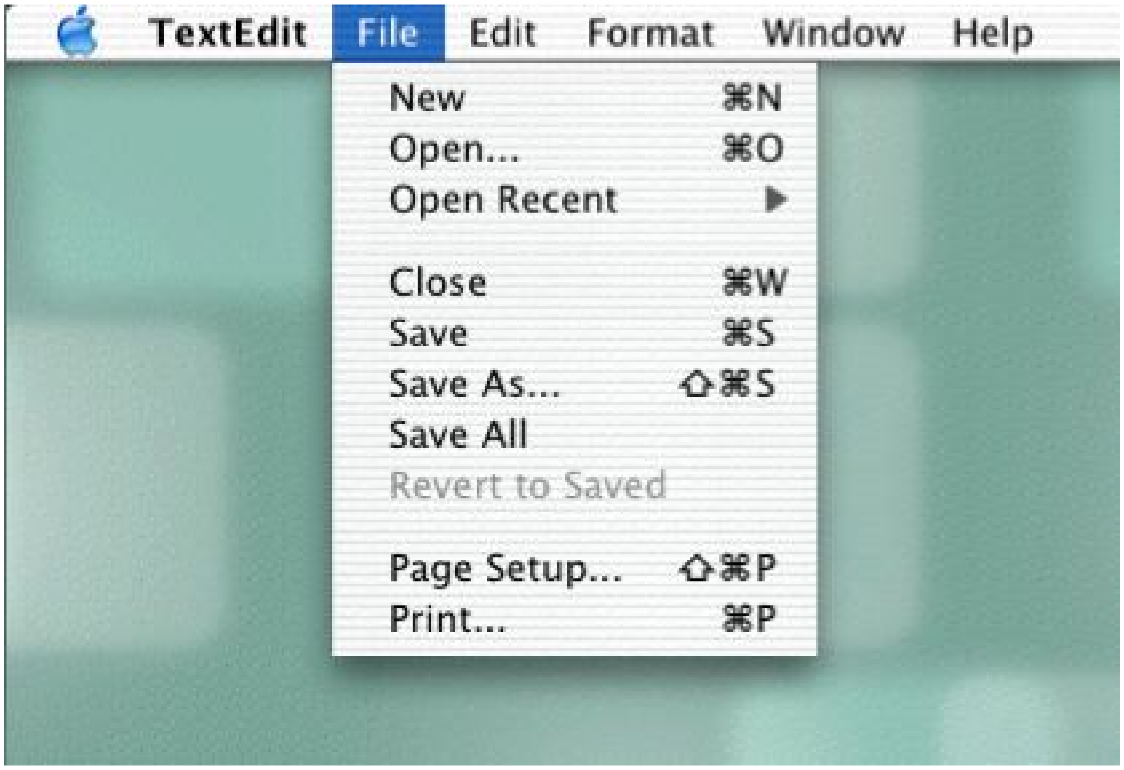
- user application
- utility
- viewer/player/accessory
- document
- preference
- plug-in
- hardware
- removable media

In addition, Apple encourages the use of *icon families*--that is, icons that are associated with a given application. Each application should have, in addition to the icon for the application itself, icons for some or all the following categories: document, preferences, plug-in, and perhaps other application-specific icons. All the icons in an icon family should conform to certain guidelines (for example, document icons are rectangular with the upper right corner curling forward in a triangle shape), and they should all be visually related through some common graphic element.

For further information regarding icons, including important behaviors to implement, see the Icons chapter of *Apple Human Interface Guidelines*..

Menus

One seemingly great difference between the Windows and Mac OS platforms is the latter's use of a single menu bar at the top of the primary display, instead of menu bars at the top of each application window. Though the effect is visually different, there is very little difference functionally. On both platforms, users see a menu bar that is associated with the active document--the only difference is where that menu bar is located on the desktop. In Mac OS X, when the user switches to a document governed by a different application, the system automatically displays the menu bar (and the menus' contents) of the newly active application.



Programmatically, you create your menus and menu items under Mac OS X the same way you do under Windows, using Interface Builder or another IDE.

Here are some notes about individual menus within Mac OS X:

- Because a Mac OS X window does not have a menu bar, it doesn't have a document menu or an application menu (the icon-based menu on the left side of a document's or MDI application's menu bar, respectively). Many of the functions on these menus are contained in the Apple and Application menus (see below).

- The Apple menu, denoted by the blue Apple logo on the left side of the menu bar, is a system-wide menu available at all times. The menu is defined by Apple, and you cannot modify it.
- The Application menu appears immediately to the right of the Apple menu. Its name is the name of the currently active application. (For example, if TextEdit is the active application, the menu immediately to the right of the Apple menu is named "TextEdit".) The application menu contains menu items for your application's About window and Preferences dialog, as well as the Services and Quit menu items. Your application may choose to use system-wide services made available from the Services submenu, or it may provide its own services to other applications. For more details on this, see *Inside Mac OS X: Setting up Your Application to Use the Services Menu*.
- You should include File, Edit, Window, and Help menus in your application. In general, your application-specific menus should appear on the right hand of the menu bar after the Window menu.
- At the extreme right end of the menu bar, Apple embeds several non-menu items called *menu bar status items*. These items relate to hardware and network settings. You should not place any status items in this area; instead, you can attach them to your application's tile in the Dock.

For further information regarding menus, see the Windows chapter of *Apple Human Interface Guidelines*.

Windows

It is generally accepted that multiple window types help users visually distinguish a window's use by its appearance. Mac OS X uses the following kinds of windows:

- document windows
- drawers
- utility windows (palettes)
- dialogs

Dialogs are, technically speaking, windows. However, because their function is radically different from the other kind of windows, they are discussed separately.

There are a number of well defined recommended behaviors for Mac OS X windows, and users expect to see them. Some of these defined behaviors are provided automatically by the operating system itself--for example, the behaviors that define the movement and resizing of windows. You must implement certain behaviors and, in some cases, decide whether they should be implemented. Here are some examples:

- the size and placement of new windows
- the conditions under which scroll bars appear, and their scrolling behavior
- automatic scrolling of window content under certain conditions
- click-through (the behavior that results from clicking a control in an inactive window)
- the behavior of the zoom button

For further information regarding windows and drawers, see the Windows chapter of *Apple Human Interface Guidelines*.

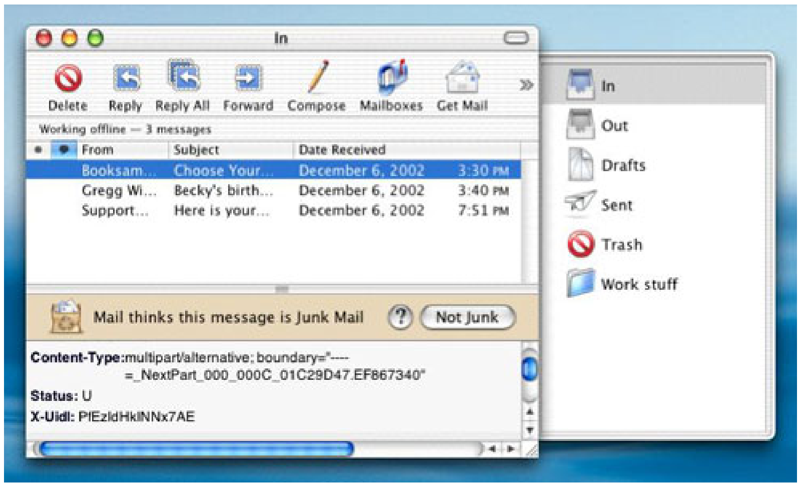
Document Windows

Document windows provide a visual interpretation of file-based user data. If your application does not have documents as such, you will probably use a document window to represent the application itself.

Drawers

A drawer is a child window that slides out from a parent window; the user can open or close it while the parent window is open. You usually employ a drawer to display frequently-accessed controls that don't need to be visible all the time. (If controls do need to be visible all the time, you should place them in a utility window or a Mac OS X-style toolbar.)

To see a good example of a drawer, open the Mac OS X Mail application (found in the Applications folder). Click the Mailboxes icon at the top of the Mail window to see the drawer, which contains the icons for the In, Out, and Trash mailboxes, as well as icons for user-created mailboxes.

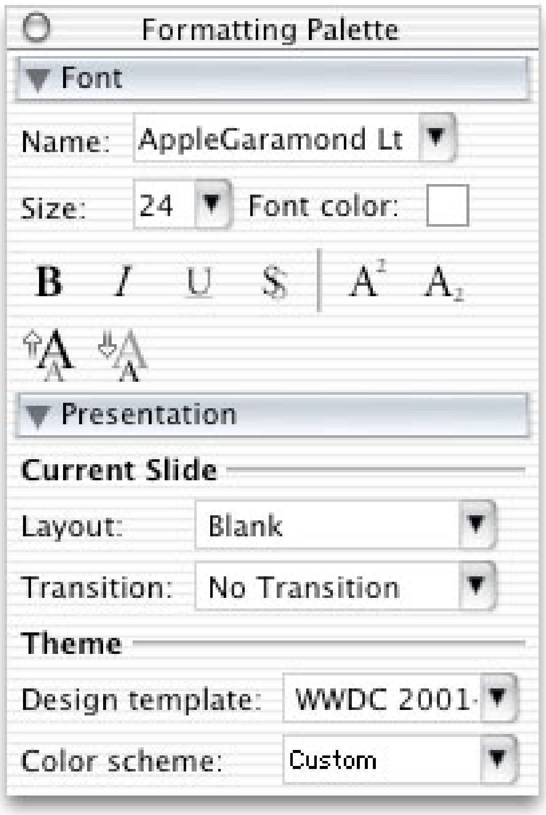


From	Subject	Date Received	
Booksam...	Choose Your...	December 6, 2002	3:30 PM
Gregg Wi...	Becky's birth...	December 6, 2002	3:40 PM
Support...	Here is your...	December 6, 2002	7:51 PM

```
Content-Type:multipart/alternative; boundary="----
        =_NextPart_000_000C_01C29D47.EF867340"
Status: U
X-Uidl: PfEzldHklNNx7AE
```

Utility Windows

Utility windows (also called palettes) float above document windows; they most often provide tools, controls, or settings that affect the active document window. You are responsible for creating and maintaining them.



Mac OS X utility windows differ from their Windows counterparts in several significant ways:

- They have a drag strip that enables users to reposition them on the desktop, but the drag strip usually does not have a text title.
- When your application is in the background (that is, its windows are inactive but visible), its utility windows should be hidden from view. (You are responsible for implementing this behavior.)
- In addition to application-specific utility windows, Mac OS X also supports system-wide utility windows, which remain visible even when the parent application is hidden from view.

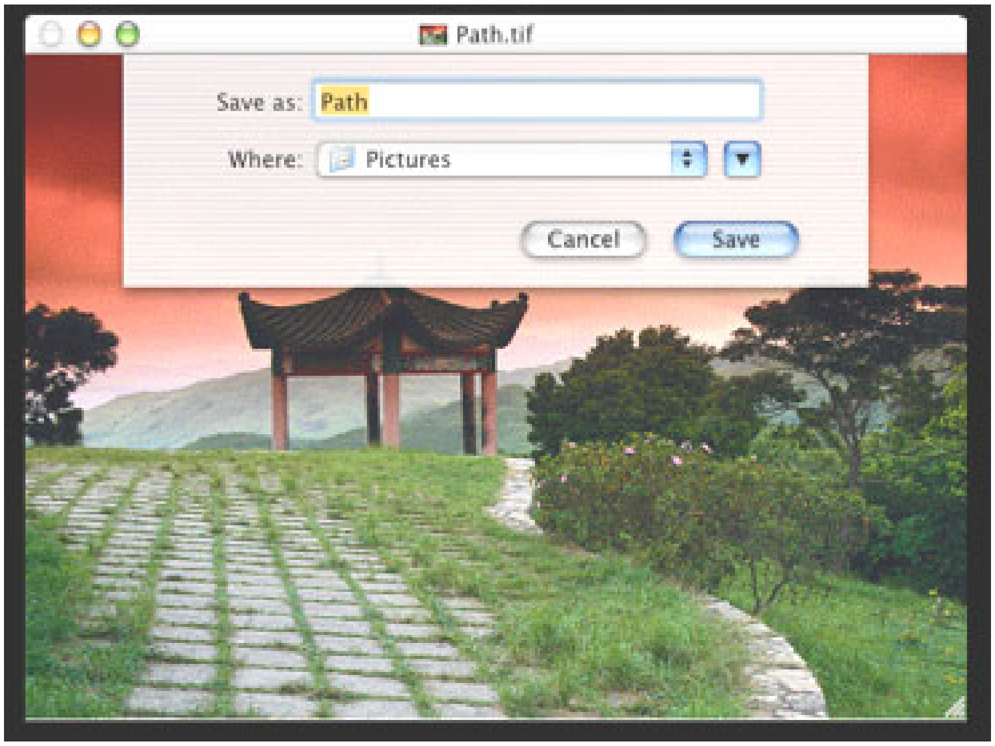
Dialogs

Dialogs are windows that your application uses to get information from the user. Dialogs that present information to the user but accept no information back are called *alerts* (in Windows terminology, message boxes). The Windows platform can display two kinds of dialogs, modeless and modal. Mac OS X distinguishes between two different kinds of modal dialogs:

- *Application-modal dialogs*. These are similar to Windows modal dialogs in that no work can be accomplished in the application before this dialog is closed.

- *Document-modal dialogs*. These dialogs must be closed before the user can continue working in the document to which the dialog is attached. The user can, however, work in other documents belonging to the same application.

Document-modal dialogs have a distinctive physical appearance; in Apple terminology, they are called *sheets*. A sheet appears to grow out from the document window's title bar, and when it is closed, it disappears in the same way.



For further information regarding dialogs and sheets, including important behaviors to implement, see the Dialogs chapter of *Apple Human Interface Guidelines*.

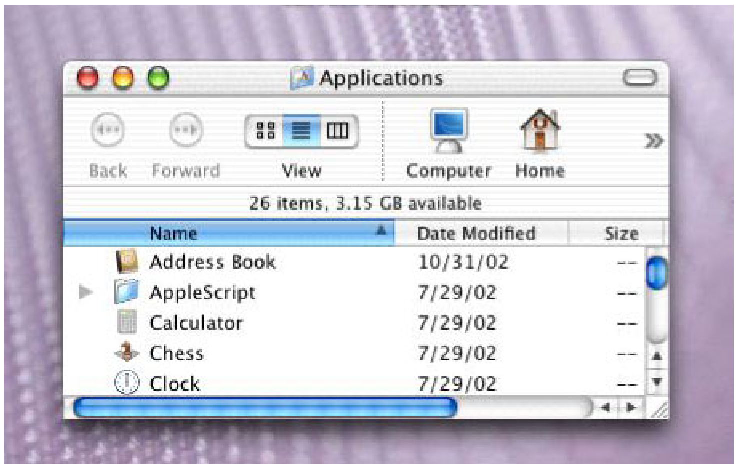
Browsing the file system and saving files are two important user activities, and Mac OS X provides the Open and Save dialogs for this purpose. You will use the Navigation Services API to create the open and save dialogs for your application. As of this writing, the current version of the Navigation Services API is 3.0, so be sure to ignore older Apple documentation about the earlier versions of Navigation Services. Mac OS X also provides dialogs that assist the user in choosing a printer, setting up a print job, and printing the document.

Controls

As stated earlier, Mac OS X includes a versatile set of controls for building your user interface. Most of the controls you are familiar with have equivalents in Mac OS X. In those cases where there are no equivalents (see Table 2, above), there are still ways to provide equivalent behavior when you port your application to Mac OS X.

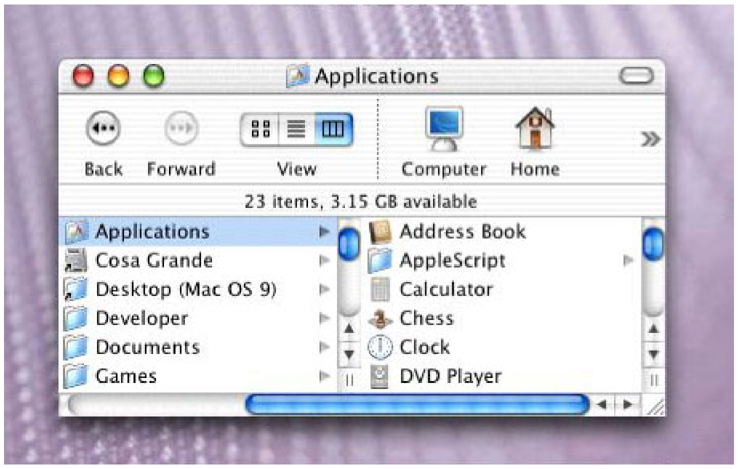
Beginning with Mac OS X version 10.2 (Jaguar), Apple began using a new API for drawing controls called HIView. This method, which is similar to the Win32 implementation of controls as child windows, largely replaces the older Control Manager API used in earlier versions of the Mac OS. It should be noted that you will still need to use the Control Manager API for certain controls--for example, the list box, scrolling text field, and data browser controls.

HIView is an object-oriented API that offers a number of advantages over the Control Manager. Through a process called *compositing*, HIView can draw complex controls faster. In addition, you can subclass HIView controls to create controls with custom behavior. HIView is a Quartz 2D-based API, but you can still use it along with QuickDraw calls.



Name	Date Modified	Size
Address Book	10/31/02	--
AppleScript	7/29/02	--
Calculator	7/29/02	--
Chess	7/29/02	--
Clock	7/29/02	--

You should become familiar with a very useful and versatile Mac OS X control that is not available in Windows, the *data browser*. It can take two forms: a list view (shown above, for displaying and changing data arranged in rows and columns) and a column view (shown below, which Mac OS X uses as one of its ways of displaying a hierarchy of files in the Finder).



The Mouse

If you have designed your product to work with a two-button mouse, then you may have concerns about the one-button mouse that ships with every Macintosh computer. Mac OS X provides complete support for multi-button mouse devices, as well as for computer mice that include a scroll wheel.

It is true that every Mac ships with a one button mouse; but, this is not a problem that you will need to work around. Many Mac users know that you can display the Mac's contextual menu (a pop-up menu) by holding down the Control key and clicking the mouse button. Mac OS X was designed so that the default behavior for clicking a mouse's right button is to display the contextual menu. This means that your Win32 application's right mouse button behavior maps quite naturally to Mac OS X. Because some users do not use contextual menus, Apple strongly recommends that all menu items that appear from clicking the right and middle mouse buttons also be available somewhere within your application's menus.

Some technical notes: Mac OS X supports three-button mice through the `kEventParamMouseButton` parameter, which is delivered as part of a mouse-down or mouse-up event. Also, Mac OS X supports the scroll wheel through the `kEventMouseWheelMoved` event, which is sent to the system event queue whenever the mouse wheel is moved. See the "Mouse Events" section of *Inside Mac OS X: Handling Carbon Events* for details.

The Keyboard

Apple has traditionally emphasized direct manipulation over typing, with the result that Mac OS users do not expect to see keyboard equivalents (shortcuts) for every menu item. For those who find it difficult to use the mouse, Apple has included a Full Keyboard Access option, which enables users to access menus, controls, and the Dock from the keyboard.

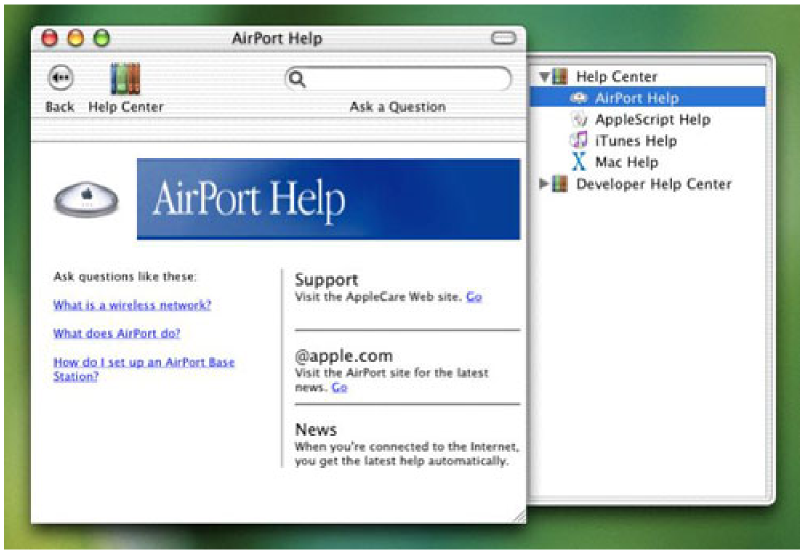
Mac OS X keyboard equivalents are characterized by their use of the Command key, an extra key on the keyboards of Apple computers that has no equivalent on Windows computers. (Apple keyboards also have a Control key, as well as an Option key that doubles as an Alt key.) Apple specifies standard keyboard equivalents, most often the Command key and a single unshifted letter key, for menu items common to most or all applications (for example, Command plus the letters z, x, c, and v for the undo, cut, copy, and paste operations). Mac OS X indicates that a menu item has a shortcut by listing the shortcut name, flush right, after the text of the menu item. (See the screen shot in the "Menu" section, above.)

You can find Apple's lists of reserved and recommended keyboard equivalents in the "Keyboard Equivalents" section of *Inside Mac OS X: Aqua Human Interface Guidelines*. If you want to add your own keyboard equivalents for operations not in the list of reserved and recommended keyboard equivalents, Apple advises that you do so only for operations that users are likely to use frequently.

Few Mac OS X applications make use of function keys, and most Mac users don't associate function keys with specific system-wide behavior (as opposed to Windows users, who likely associate the F1 key with invoking an application's help function). For these reasons, you should not map any functions to the F1 through F12 keys. Instead, create keyboard shortcuts that use the Command key in combination with other non-function keys.

Apple Help

Apple's philosophy of help is that users turn to it when all else has failed. They know what they want to do but not how to do it, and they are frustrated. You should make your help system task-oriented, so that users can quickly find the help they need. Also, by limiting each help page to covering one simple task and writing descriptive help-page titles, you can increase the likelihood that your users will be able to find exactly the right help page.



Ask questions like these:

What is a wireless network?

What does AirPort do?

How do I set up an AirPort Base Station?

Support
Visit the AppleCare Web site. Go

@apple.com
Visit the AirPort site for the latest news. Go

News
When you're connected to the Internet, you get the latest help automatically.

How Users Get Help

The Help menu is an important part of the Mac OS X user interface. When you write your help documentation (in a format called a *help book*), Mac OS X automatically adds the Help menu and, under it, a menu item named "<your-application-name> Help". When a user selects this menu item, Mac OS X opens your help documentation.

There are two ways that you can invoke your help function from within the application itself. First, you can design your application so that, when users point at certain interface elements and invoke the contextual menu, the top menu item opens a selected page of your help book. Second, you can add a Help button to one of your application's windows or dialogs; when the user clicks this button, the system displays a selected page from your help book.

Help Viewer

Help Viewer is the part of Apple Help that displays help books, which contain help documentation implemented as HTML 3.2-compliant web pages. In addition to displaying help books, Help Viewer also provides a built-in search facility that helps users find specific help pages and returns a list of possible help topics of interest sorted by relevance.

Apple Help makes it possible for you to offer your customers a rich, friendly experience when they need help using your application. To see Apple Help at its best, examine the opening help pages of the applications that ship with Mac OS X. You will see that they present a sophisticated and yet friendly introduction to each application's help function. In particular, from the Finder, invoke the Mac Help menu item from the Help menu and explore the help documentation for Mac OS X. After that, click the Help Center icon and invoke the help documentation for AppleScript, AirPort, iMovie, and iTunes. Each has a different personality and structure; all of them contain ideas you may want to adopt when you design your help documentation.

Creating Your Help Book

There are five steps that you must take to create the help book for your application:

1. Design the structure of your help book.
2. Write the help pages in HTML.
3. Add enhancements to improve the usefulness of your help book.
4. Index your help book using the Apple Help Indexing Tool.
5. Register your help book to make it part of the Mac OS X help system, and place its files in the correct location.

You will find details on how to do these things in *Providing User Assistance with Apple Help*.

Enhancing Your Help Book

There are several ways you can enhance your help book to make it more useful. The Help Viewer adds the following non-HTML capabilities:

- launching another application from a hyperlink in your help text
- displaying pictures, sounds, and movies using QuickTime
- adding "do this for me" actions to your help pages using the AppleScript scripting language

You can add the following items to make your help book more useful:

- *HTML anchor tags*: These make it possible for you to open Apple Help to a specific location under program control.
- *Description tags*: When Apple Help displays the result of the search, these tags give additional information about each help item found.
- *Keywords*: You can specify lists of keywords that are synonyms for each other. Help Viewer broadens its search by adding synonyms to the search terms that the user specifies. This increases the accuracy of users' searches, since they will be directed to the right help topic regardless of which synonym they use when phrasing their query.
- *ROBOTS meta tags*: You can add this tag to influence how the Apple Help Indexing Tool indexes different parts of your help book. Doing so will result in an index that returns better search results.
- *Help URLs*: These are special URLs, visible to the user as hyperlinks, that enable users to open other help books, HTML pages, or text files. In addition, you can create help URLs that open the main Apple Help Center page or that automatically run a search query using arguments that you specify.

Finally, you can enhance your help book by maintaining certain pages on a remote server anywhere on the Internet. In this way, you can keep critical information updated, while still keeping it a part of your application's help function. (Of course, this assumes that users have Internet access.) See "Preparing an Index for Remote Pages" in the book *Providing User Assistance with Apple Help* for details.

Miscellaneous Topics

Here are some miscellaneous items relating to the user interface:

- Ease of use, customization and WYSIWYG (what you see is what you get) are key parts of the Mac OS X user experience. Mac users expect objects on the desktop to behave in predictable and consistent ways. In particular, they expect to be able to change the names and locations of applications and folders without that causing, directly or indirectly, anything on their computer to stop working properly. For this reason, your application will be expected to recover gracefully if users make such changes.
- In some cases, the structure of Mac OS X isolates your application from the effects of user-initiated changes; for example, changing the name of an application icon or enclosing folder does not prevent it from working. In other cases, you will need to design robustness into your application. If possible, you should avoid hard-coding absolute path names into your application. Possible alternatives include using relative path names, checking multiple places for a given file, and asking the user to find a lost file (and then remembering the new location).
- The correct place to store application-specific and user-specific settings that need to be remembered is in a preferences property list, which is maintained by Mac OS X. Using the routines described in `CFPreferences.h`, you can set and retrieve such values.
- A good way to store pointers and other data related to a given window is to use the `SetWindowProperty` and `GetWindowProperty` routines to attach any reasonable number of values to that window. You can find these routines by looking in the `MacWindows.h` file within the HIToolbox framework (use the UNIX `locate` command to find this file).
- Mac OS X applications will be used in international locations, and your application should be "language-blind"-- that is, it should not know or care what language it is using to displaying its information. Users specify, in order of decreasing preference, the language they want to see used in menus and dialogs, and Mac OS X automatically uses whatever available language resource best matches those preferences. If you create your application according to Mac OS X guidelines, your application will automatically display menu and dialog text in the preferred language without you having to keep track of what languages are available or what language the user prefers. Part of implementing this behavior involves placing all user-visible strings in a file called `Localizable.strings`, as described in the "Internationalization" chapter of *Inside Mac OS X: System Overview*.
- Mac OS X does not have a technology for embedding software objects from different applications into the same document (such as Microsoft's OLE). If your Win32 application uses OLE, you will need to decide whether or not to provide similar capabilities in your Mac OS X version and, if so, how to implement them. From a user interface perspective, the lack of such a facility in Mac OS X simplifies both the user interface and the code underneath it.
- Apple's user interface guidelines discourage the use of group boxes (rectangles used to group related controls). Rather, you should use space or separator lines to achieve the same effect, but with less visual clutter.
- Mac OS X provides two standard ways to abort an operation using the keyboard: pressing the Escape key, and pressing the Command-. (period) key combination. Both methods should invoke the same functionality.

For Further Information

Your main reference for Mac OS X user interface issues is *Inside Mac OS X: Aqua Human Interface Guidelines*. In addition to the links to this book, you may find some of the other links below helpful.

Apple Human Interface Guidelines	*Apple Human Interface Guidelines*
A Quick Tour of XCode: Designing a User Interface	*A Tour of Xcode*
Key User Experience Differences between Windows and Mac OS X	http://developer.apple.com/ue/switch/windows.html
Apple User Experience page	http://developer.apple.com/ue/
Apple Aqua page	http://developer.apple.com/ue/aqua/
links to documentation on Apple Help, including *Providing User Assistance with Apple Help*	*Apple Help Reference*
Introducing HIView	*HIView Programming Guide*
HIView reference documentation	*HIView Reference*
Event Manager Reference	*Event Manager Reference*
Handling Carbon Windows and Controls	*Handling Carbon Windows and Controls*
Navigation Services (Open and Save dialogs)	http://developer.apple.com/documentation/Carbon/Reference/Carbon_Spec_Porting/Navigation_Services.html
List Manager documentation	http://developer.apple.com/documentation/Carbon/Reference/Carbon_Spec_Porting/List_Manager.html
Data Browser Control technical note	http://developer.apple.com/technotes/tn/tn2009.html
preferences property list	located on a Mac OS X hard disk at `/System/Library/Frameworks/CoreFoundation.framework/Headers/CFPreferences.h`
Inside Mac OS X: Setting up Your Carbon Application to Use the Services Menu	*Setting Up Your Carbon Application to Use the Services Menu*
Developer Tools page	http://developer.apple.com/tools/index.html

Document Revision History

This table describes the changes to *Porting to Mac OS X from Windows Win32 API*.

Date	Notes
2009-05-06	Corrected typos.
2006-01-10	Replaced references to Project Builder with references to Xcode.
2005-08-11	Fixed broken link.
2005-04-29	Changed "Rendezvous" to "Bonjour."
2004-08-31	Fixed a bunch of dead links.
2004-02-11	Converted *Porting to Mac OS X from Windows Win32 API* to standard document format from HTML source.