INSIDE MACINTOSH

Programming With the Mac OS 8.5 Window Manager

March 8, 1999
Technical Publications

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

About the Mac OS 8.5 Window Manager

Contents

Mac OS applications typically interact with users via windows on the screen. You can use the Window Manager to create, display, and manage the drawing and behavior of windows.

This document describes the Window Manager application programming interface (API) introduced with Mac OS 8.5 and Appearance Manager 1.1. Preexisting Window Manager functionality is not discussed in this document. For a description of the Mac OS 8 Window Manager API, see *Mac OS 8 Window Manager Reference*. For descriptions of the pre–Mac OS 8 Window Manager API, see *Inside Macintosh: Macintosh Toolbox Essentials*.

See the following sections for descriptions of various features of the Mac OS 8.5 Window Manager.

Window Creation, Storage, and Disposal

Prior to the Mac OS 8.5 Window Manager, there were two forms of window data: the window structure in memory that is referenced at execution time by a window pointer and the window resource (of type `'WIND'`). With the Mac OS 8.5 Window Manager, there are three forms of window data from which your application can create a window: the live window, the window's collection data, and the window's flattened collection data. See "Creating a Window" (page 32)

for an example of how your application might use Mac OS 8.5 Window Manager functions to create a window.

A **collection** is an abstract data type, defined by the Collection Manager, that allows you to store multiple pieces of related information. For purposes of the Window Manager, however, a collection might best be understood as an intermediate state between a live window and a `'wind'` resource. Using the Mac OS 8.5 Window Manager, your application can store any window, even those not created with Mac OS 8.5 Window Manager functions, into a collection. You can also store data associated with the window into the same collection. This provides a quick way for your application to save a simple document.

From a collection, your application can create a **flattened collection**—that is, a stream of address-independent data—using the Collection Manager. Because the `'wind'` resource consists of an extensible flattened collection, your application can store a flattened collection consisting of a window and its data into a `'wind'` resource using the Resource Manager. “Storing a Document Window Into a Collection” (page 57) provides an example of how your application might store a window and its data as a single flattened collection in an extended `'wind'` resource.

The Mac OS 8.5 Window Manager provides the following functions to create and store windows:

- `CreateNewWindow` (page 67) creates a window from parameter data.
- `CreateWindowFromResource` (page 69) creates a window from `'wind'` resource data.
- `CreateWindowFromCollection` (page 68) creates a window from collection data.
- `StoreWindowIntoCollection` (page 70) stores data describing a window into a collection.

With the Mac OS 8.5 Window Manager, all references to a window are counted and thereby tracked. As there is only one owner with a reference to a given window when it is first created, windows are created with a **reference count** (or “owner count”) of one. When another owner acquires a reference to a window, the window’s reference count increases by one. When an owner stops using a window and releases its reference, the number of references to the window decreases by one. When the reference count reaches zero, the Window Manager automatically disposes of the window.

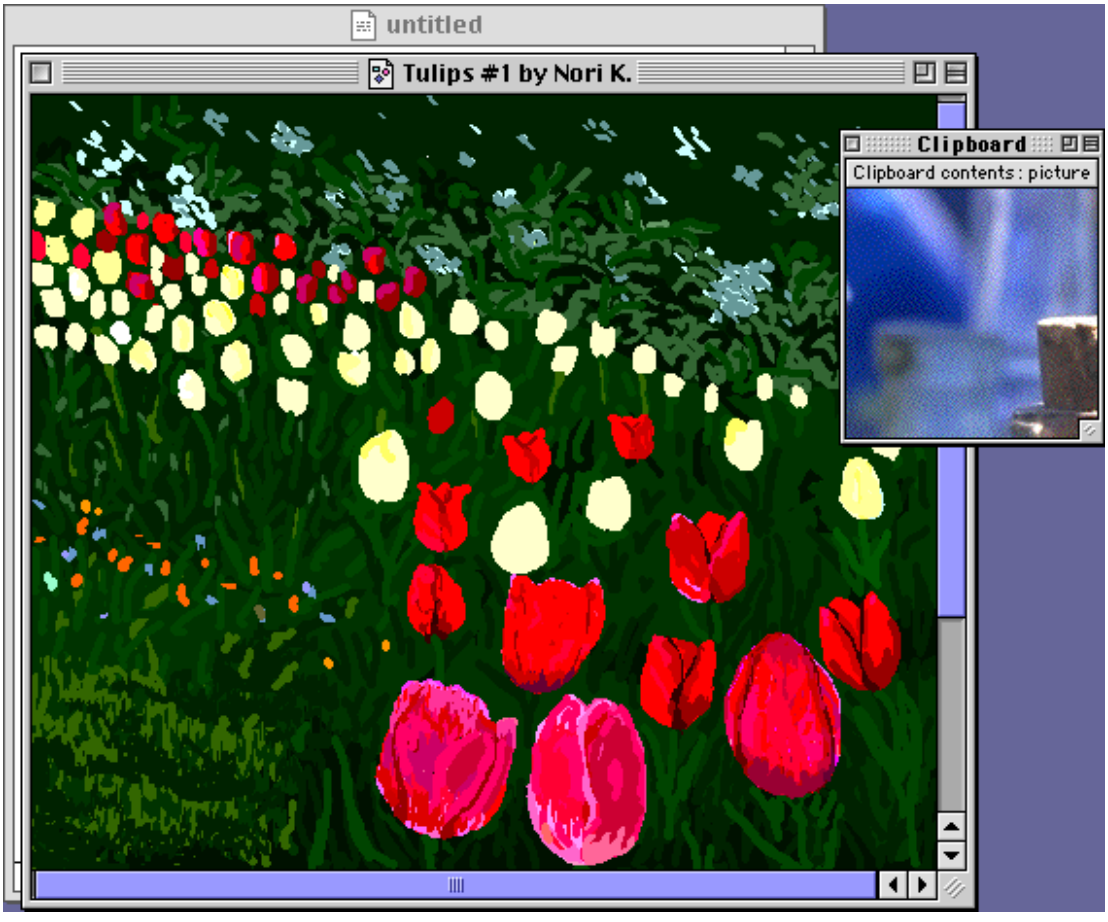
The Mac OS 8.5 Window Manager provides the following functions for working with references to windows:

- `GetWindowOwnerCount` (page 72) obtains the number of existing references to a window.
- `CloneWindow` (page 71) increments the number of references to a window.

Floating Windows

Windows are often placed on the display screen so that one window appears to be behind another. This visual overlapping gives the user an impression of depth. A **floating window** is so-named because its front-to-back display order (that is, its z-order placement relative to other windows on the screen) makes it appear to float in front of document windows. In Figure 1-1, the “Clipboard” window appears to float in front of the active and inactive document windows.

Figure 1-1 Floating windows



Because earlier versions of the Window Manager defined only the look of floating windows, not their floating behavior, some applications contain code that implements a floating effect for tool palettes and other such windows. However, your application can now use the Window Manager to automatically sort floating and non-floating windows into separately z-ordered groups, thereby enforcing the proper front-to-back display order.

IMPORTANT

Floating windows are supported under Mac OS 8.6 and later. ▲

The Mac OS 8.5 Window Manager provides the following functions for displaying floating windows:

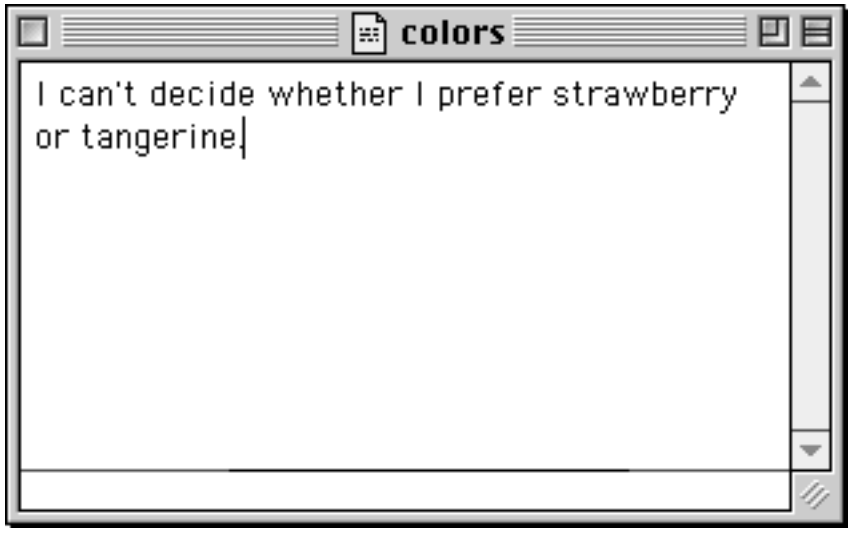
- `InitFloatingWindows` (page 75) initializes the Window Manager and sorts your application's windows into the proper front-to-back display order.
- `ShowFloatingWindows` (page 77) shows an application's floating windows.
- `HideFloatingWindows` (page 74) hides an application's floating windows.
- `AreFloatingWindowsVisible` (page 73) indicates whether an application's floating windows are currently visible.

Window Proxy Icons

The Mac OS 8.5 Window Manager supports the display of a small icon in the title bar of document windows (next to the window title) that serves as a proxy for the document's icon in the Finder. This **proxy icon** appears and behaves the way the icon for the document does in the Finder. For example, the user can drag a document's proxy icon to move or copy the document file.

Additionally, the proxy icon is a source of visual feedback for the user on the current state of the document, such as whether the document window is a valid drag-and-drop target and whether the document has unsaved changes. Figure 1-2 shows a proxy icon in a window that contains a document with no unsaved changes.

Figure 1-2 Proxy icon in a window containing a document with no unsaved changes



The Mac OS 8.5 Window Manager provides the following functions for establishing proxy icons in your application's windows. See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

- `SetWindowProxyFSSpec` (page 104) associates a file with a window.
- `GetWindowProxyFSSpec` (page 99) obtains a file system specification structure for the file that is associated with a window.
- `SetWindowProxyAlias` (page 101) associates a file with a window.
- `GetWindowProxyAlias` (page 98) obtains an alias for the file that is associated with a window.
- `SetWindowProxyCreatorAndType` (page 102) sets the proxy icon for a window that lacks an associated file.
- `SetWindowProxyIcon` (page 105) overrides the default proxy icon for a window.
- `GetWindowProxyIcon` (page 100) obtains a window's proxy icon.
- `RemoveWindowProxy` (page 100) dissociates a file from a window.

Note that, in Figure 1-2, the proxy icon is drawn in the enabled state to indicate that the file represented by the icon has no unsaved changes and that the user may therefore manipulate the icon and thereby the file itself. If a user drags a proxy icon to a folder, Finder window, the desktop, or another volume, the file represented by the proxy icon is moved or copied accordingly, as if the user had dragged the file's icon in the Finder.

The Mac OS 8.5 Window Manager provides the following functions for dragging proxy icons. See "Tracking a Window Proxy Icon Drag" (page 45) for an example of how your application can call these functions.

- `TrackWindowProxyDrag` (page 112) handles all aspects of the drag process when the user drags a proxy icon.
- `TrackWindowProxyFromExistingDrag` (page 113) allows custom handling of the drag process when the user drags a proxy icon.
- `BeginWindowProxyDrag` (page 107) creates the drag reference and the drag image when the user drags a proxy icon.
- `EndWindowProxyDrag` (page 108) disposes of the drag reference when the user completes the drag of a proxy icon.

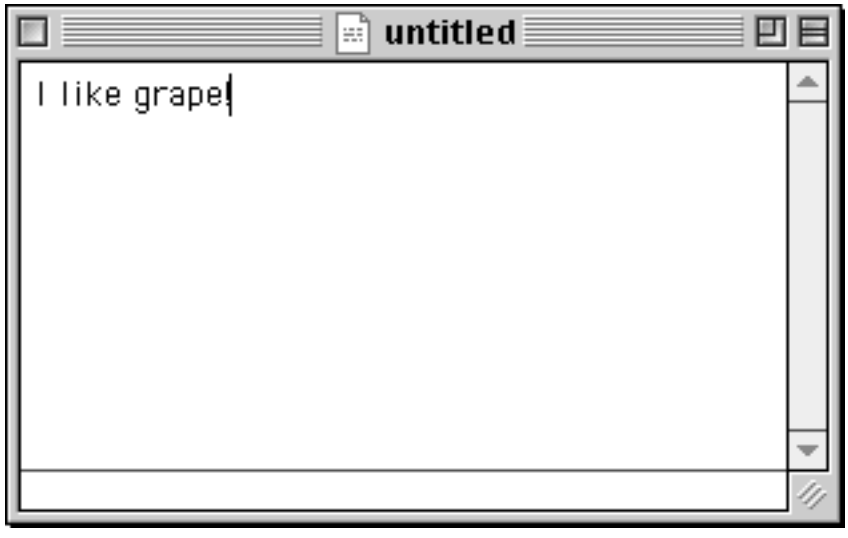
An application typically tracks the modification state of a document. A common reason to do so is to inform the user that they have made changes to the document which they might wish to save before closing the window.

When your application uses proxy icons, it should inform the Window Manager when a document has unsaved changes. When you do so, the Window Manager displays the document's proxy icon in a disabled state and prevents the user from dragging the proxy icon. Disabled proxy icons cannot be dragged because unsaved documents cannot be moved or copied in a manner predictable to the user. Figure 1-3 shows a proxy icon in a document window with unsaved changes.

IMPORTANT

The only time that a document's proxy icon should be disabled is when the document has unsaved changes. Your application should not disable the proxy icon at any other time. ▲

Figure 1-3 Proxy icon in a window containing a document with unsaved changes

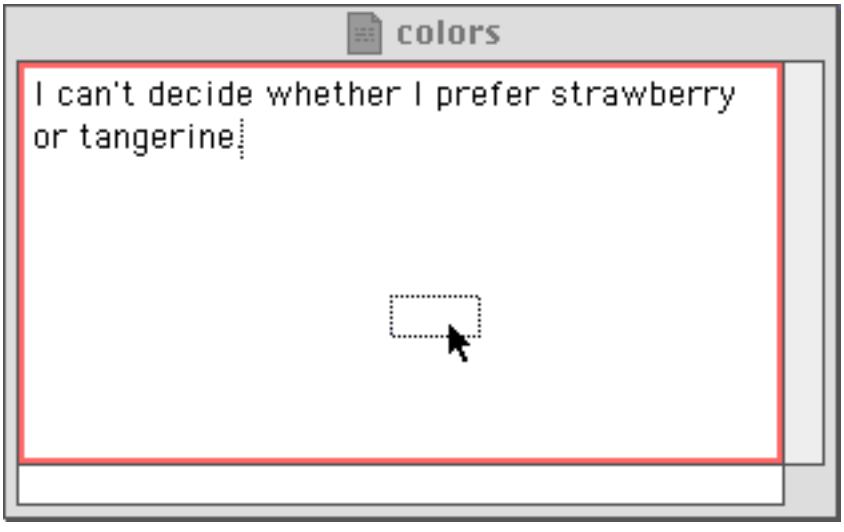


The Mac OS 8.5 Window Manager provides the following functions for accessing the modification state of a window. See “Setting a Window’s Modification State” (page 57) for an example of how your application can call these functions.

- `SetWindowModified` (page 111) sets the modification state of the specified window.
- `IsWindowModified` (page 111) obtains the modification state of the specified window.

When the user drags content that an application can accept into the content area of one of its windows, the structure region of the window, including the proxy icon, should become highlighted, as shown in Figure 1-4. This gives visual feedback that the window is a valid destination for the content.

Figure 1-4 Proxy icon in a window that is a valid drag-and-drop target



The Mac OS 8.5 Window Manager provides the following function for indicating to the user whether a window is a valid drag-and-drop target:

- `HiliteWindowFrameForDrag` (page 110) sets the highlight state of the window's structure region to reflect the window's validity as a drag-and-drop destination.

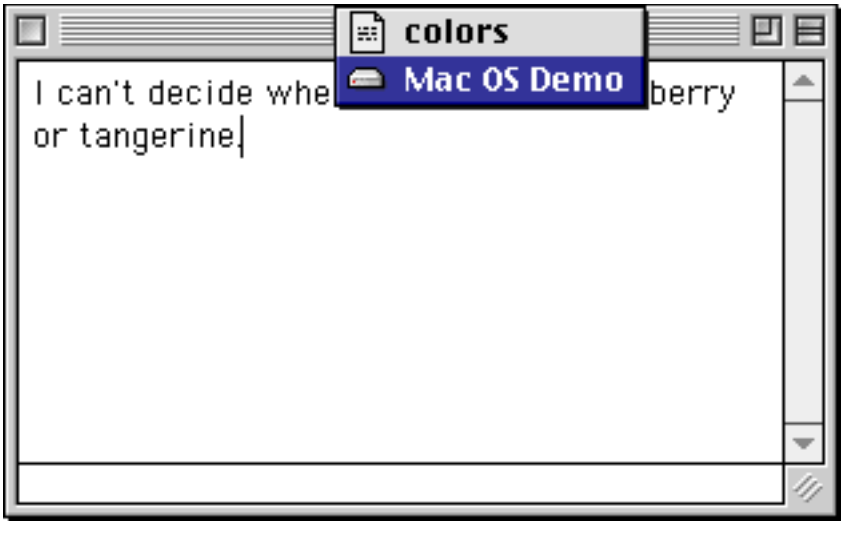
Figure 1-5 compares the various states of a proxy icon: enabled, for a document with no unsaved changes; disabled, for a document that does have unsaved changes; and highlighted, for when the document window is a valid destination for content that the user is dragging.

Figure 1-5 Proxy icon states

Window Path Pop-Up Menus

The Mac OS 8.5 Window Manager provides system support for your application to display window path pop-up menus—like those used in Finder windows. If your application uses window path pop-up menus, when the user presses the Command key and clicks the window title, your window displays a pop-up menu containing a standard file system path. The window path pop-up menu informs the user of the location of the document displayed in the window and allows the user to open windows for folders along the path. Figure 1-6 shows a window path pop-up menu for a document window.

Figure 1-6 Window path pop-up menu



The Mac OS 8.5 Window Manager provides the following functions for handling the activation of window path pop-up menus. “Displaying a Window Path Pop-Up Menu” (page 46) shows how your application can handle a user request to display the window path pop-up menu.

- `IsWindowPathSelectClick` (page 115) reports whether a mouse click should activate the window path pop-up menu.
- `WindowPathSelect` (page 116) displays a window path pop-up menu.

Transitional Window Animations and Sounds

Prior to Mac OS 8.5, the Window Manager supported playing a sound to accompany the transitional "window shade" animation that occurs when a user clicks the collapse box of a window. In addition to this combination of animation and sound for a user interaction with a window, the Mac OS 8.5 Window Manager now supports a combination of animation and sound to go with the opening and closing of windows.

The Mac OS 8.5 Window Manager provides the following function for displaying a window with animation and sound:

- `TransitionWindow` (page 78) displays an animation and plays the theme-appropriate sound for a window when it is shown or hidden.

"Creating a Window" (page 32) provides an example of how your application might call the `TransitionWindow` function.

Window Zooming

When the user clicks a window's zoom box, a window zooms between two states, the user state and the standard state. The **user state** is any size and position in which the user can place the window on the desktop. The **standard state** is the size and position that the application defines as being best for the display of the data contained in the window. There are human interface guidelines for how best to determine a window's standard state, based upon its current user state, but prior to Mac OS 8.5 there were no system-supplied functions that enforced these guidelines for your application.

When you use the Mac OS 8.5 Window Manager zooming functions, your application automatically conforms to the human interface guidelines for determining a window's standard state, as follow:

- A window should move as little as possible when zooming between the user state and standard state, to avoid distracting the user.
- A window in its standard state should be positioned so that it is entirely on one screen.
- If a window straddles more than one screen in the user state, when it is zoomed to the standard state it should be zoomed to the screen that contains the largest portion of the window's content region.
- If the ideal size for the standard state is larger than the destination screen, the dimensions of the standard state should be that of the destination screen, minus a few pixels' boundary. If the destination screen is the main screen, space should also be left for the menu bar.
- When a window is zoomed from the user state to the standard state, the top left corner of the window should remain anchored in place; however, if the standard state of the window cannot fit on the screen with the top left corner anchored, the window should be "nudged" so that the parts of the window in the standard state that would fall offscreen are, instead, just onscreen.

The Window Manager also ensures that the user state is tracked accurately and gives your application access to a window's user state information through the new zooming functions.

The Mac OS 8.5 Window Manager provides the following functions for zooming windows. See "Zooming a Window Gracefully" (page 54) for an example of how your application can call these functions.

- `ZoomWindowIdeal` (page 90) zooms a window in accordance with human interface guidelines.
- `IsWindowInStandardState` (page 87) determines whether a window is currently zoomed in to the user state or zoomed out to the standard state.
- `SetWindowIdealUserState` (page 89) sets the size and position of a window in its user state.
- `GetWindowIdealUserState` (page 86) obtains the size and position of a window in its user state.

Window Position and Size

With the Appearance Manager, the look of a window frame—not just its color, but its size and shape—may vary from appearance to appearance. Because the size of a window frame can vary, the total dimensions of a window (that is, the window's structure region) may also vary, causing the window's spatial relationship to the rest of the screen to change.

Additionally, the elements of a window frame may vary in their size, shape, or position. For example, some appearances may allow the window to be resized from any corner, not just the bottom right, and as a result, when the user drags the size box around the screen, the window may move on the screen and not merely change size.

Your application can best accommodate variable window dimensions by using the functions provided by the Mac OS 8.5 Window Manager to size and position your windows, rather than via constant dimensions. Using these functions allows your application to avoid maintaining its own table of window definition IDs and their various border dimensions.

The Mac OS 8.5 Window Manager provides the following functions for working with the size and position of windows. See "Positioning a Window on the Desktop" (page 35) and "Resizing a Window" (page 56) for a discussion of these functions.

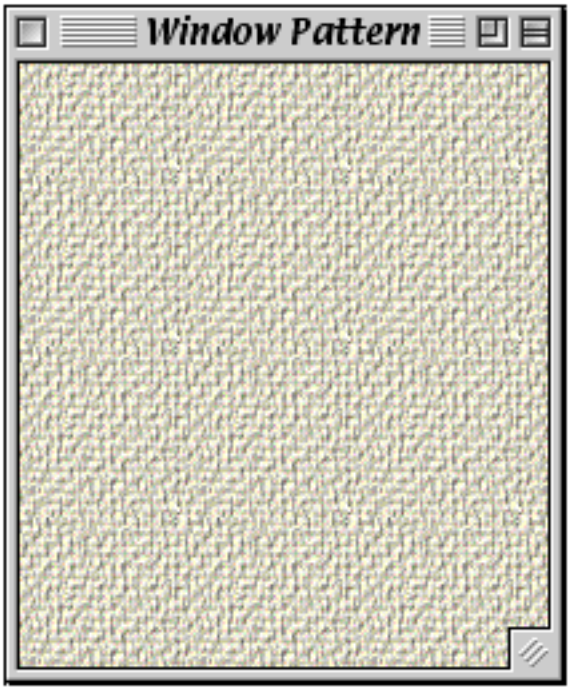
- `SetWindowBounds` (page 96) sets a window's size and position from the bounding rectangle of the specified window region.
- `GetWindowBounds` (page 92) obtains the size and position of the bounding rectangle of the specified window region.
- `MoveWindowStructure` (page 93) positions a window relative to its structure region.
- `ResizeWindow` (page 95) handles all user interaction while a window is being resized.
- `RepositionWindow` (page 94) positions a window relative to another window or a display screen.

Window Content Color

Your application and the Window Manager work together to display windows on the screen. Once you have created a window and made it visible, the Window Manager automatically draws the window's structure region (that is, its "frame") in the appropriate location. The Window Manager does not typically draw any content in a window; it only draws the color or pattern of the content region. Your application is responsible for drawing content such as text or graphics in the window's content region.

When the user exposes a window that has previously been obscured, the Window Manager redraws the exposed, invalid portions of the window. If some part of the window's content region is exposed, the Window Manager redraws it to the current content color and adds it to the window's update region. You can use the Mac OS 8.5 Window Manager to set a window's content region to a specific color or pattern, which the Window Manager then uses to redraw the content region of the window. Figure 1-7 shows an example of a window for which the content region has been set to a pattern.

Figure 1-7 A window with a patterned content area



The Mac OS 8.5 Window Manager provides the following functions for redrawing a window's content region. See "Drawing in a Window's Content Region" (page 42) for further discussion and an example of how your application might call these functions.

- `SetWindowContentColor` (page 84) sets the color to which a window's content region is redrawn.
- `GetWindowContentColor` (page 83) obtains the color to which a window's content region is redrawn.
- `SetWindowContentPattern` (page 85) sets the pattern to which a window's content region is redrawn.
- `GetWindowContentPattern` (page 83) obtains the pattern to which a window's content region is redrawn.

Window Update Regions

As the user creates, moves, resizes, and closes windows on the desktop, portions of windows may be obscured and uncovered. The Window Manager keeps track of these changes, accumulating a dynamic region known as the **update region** for each window. The update region contains all areas of a window's content region that need updating. The Event Manager periodically scans the update regions of all windows on the desktop, generating update events for windows whose update regions are not empty. When your application receives an update event, it should redraw the update region.

Both your application and the Window Manager can manipulate a window's update region. Your application can force or suppress update events by manipulating the update region, using Window Manager functions provided for this purpose. For example, in order to decrease the time that your application spends redrawing window content, you can remove an area from the update region when you know that it is in fact valid.

The Mac OS 8.5 Window Manager provides enhanced functions for manipulating the update region. They are similar to previous Window Manager functions but allow the window that you are operating upon to be explicitly specified, instead of operating on the current graphics port, so they do not require you to set the graphics port before their use. As possible, you should update pre–Mac OS 8.5 applications to use these functions rather than the pre-existing ones, so that your code more readily supports future versions of the Mac OS.

The Mac OS 8.5 Window Manager provides the following functions for updating windows. See "Maintaining the Update Region" (page 52) for a discussion of how your application can use these functions.

- `InvalWindowRect` (page 122) adds a rectangle to a window's update region.
- `ValidWindowRect` (page 124) removes a rectangle from a window's update region.
- `InvalWindowRgn` (page 123) adds a region to a window's update region.
- `ValidWindowRgn` (page 125) removes a region from a window's update region.

Data Associated With Windows

In the past, some applications have associated information with a window by creating a structure that contains both the window's window record and the application's data. However, this technique is not Carbon-compliant. Your application should use the standard mechanism provided by the Mac OS 8.5 Window Manager instead, by which any kind of data can be associated with a given window. Or, optionally, your application may use the pre–Mac OS 8.5 functionality provided by the `SetWRefCon` function, which allows your application to associate a pointer to data with a pointer to a window.

The Mac OS 8.5 Window Manager provides the following functions for associating data with windows. See "Managing Multiple Windows" (page 31) for a discussion of how your application can use these functions.

- `SetWindowProperty` (page 121) associates an arbitrary piece of data with a window.
- `GetWindowProperty` (page 118) obtains a piece of data that is associated with a window.
- `GetWindowPropertySize` (page 119) obtains the size of a piece of data that is associated with a window.
- `RemoveWindowProperty` (page 120) removes a piece of data that is associated with a window.

Window Information Accessors

The Window Manager provides accessor functions for Mac OS 8.5–related window info. Your application should always use accessor functions instead of accessing structure fields and low memory directly.

The Mac OS 8.5 Window Manager provides the following functions for determining information about windows:

- `GetWindowClass` (page 81) obtains the class of a window.
- `GetWindowAttributes` (page 80) obtains the attributes of a window.

- `FrontNonFloatingWindow` (page 80) returns a pointer to the application's frontmost visible window that is not a floating window.
- `IsValidWindowPtr` (page 82) reports whether a pointer is a valid window pointer.

Using the Mac OS 8.5 Window Manager

Contents

Macintosh applications typically use the Window Manager to simplify the display and management of windows and to retrieve basic information about user activities. Your application works with the Window Manager to present the standard user interface for windows.

This chapter discusses some programming topics for the Mac OS 8.5 Window Manager, as follows:

Managing Multiple Windows

Your application is likely to have multiple windows on the desktop at once: one or more document windows, possibly one or more dialog boxes, and possibly some special-purpose windows of your own. Only one window is active at a time, however.

You can use various strategies for keeping track of different kinds of windows. In the past, some applications have done this by creating a structure that contains both the window's window record and the application's data. However, this technique is not Carbon-compliant. Instead, you can use the

Mac OS 8.5 Window Manager function `SetWindowProperty` (page 121), which allows any kind of data to be associated with a given window. Alternately, you may use the pre–Mac OS 8.5 functionality provided by the `SetWRefCon` function, which can allow your application to associate a pointer to data with a window.

Creating a Window

You typically create a new window every time the user creates a new document, opens a previously saved document, or issues a command that triggers a dialog box.

Prior to the Mac OS 8.5 Window Manager, you could create a window in two ways:

- from window characteristics passed as parameters to the `NewCWindow` and `NewWindow` functions
- from a window resource (a resource of type `'WIND'`), with the `GetNewCWindow` and `GetNewWindow` functions

With the Mac OS 8.5 Window Manager, you can still create a window by passing parameter data, but there is an updated function for this purpose, `CreateNewWindow` (page 67), which allows you to specify Mac OS 8.5 window features. Listing 2-1 provides an example of using `CreateNewWindow` as part of an application-defined function, `MyCreateAndShowNewDocumentWindow`, that creates and displays a document window.

Because the window being created is a document window, `MyCreateAndShowNewDocumentWindow` calls the function `SetWindowProxyCreatorAndType` to establish a proxy icon for the window. See “Supporting Window Proxy Icons” (page 37) for more on working with proxy icons in your document windows.

Note that because `CreateNewWindow` creates the specified window invisibly—as do the other Mac OS 8.5 window-creation functions—`MyCreateAndShowNewDocumentWindow` also includes a call to the function `TransitionWindow` (page 78) to display the window. The `TransitionWindow` function displays an animation and plays the theme-appropriate sound for a window when it is shown or hidden. Your application may use `TransitionWindow` instead of the pre–Mac OS 8.5 Window Manager functions `ShowWindow` and `HideWindow`. Like these earlier functions, `TransitionWindow`

generates the appropriate update and active events when it shows and hides windows.

Listing 2-1 Creating and displaying a document window

```
static pascal OSStatus MyCreateAndShowNewDocumentWindow
                                (const Rect *bounds,
                                OSType fileCreator,
                                OSType fileType,
                                SInt16 vRefNum,
                                WindowPtr *window)

    {
        OSStatus err;

        // Create an invisible window

        err = CreateNewWindow ( kDocumentWindowClass,
                                kWindowStandardDocumentAttributes,
                                bounds,
                                window);

        if ( err == noErr )
        {
            // Since this is a document window, give it a proxy icon

            err = SetWindowProxyCreatorAndType (*window,
                                                fileCreator,
                                                fileType,
                                                vRefNum);

            // Make the window visible (with animation and sound)

            if ( err == noErr )
            {
                err = TransitionWindow (    *window,
                                            kWindowZoomTransitionEffect,
                                            kWindowShowTransitionAction,
                                            nil);
            }
```

```
            // Destroy the window if TransitionWindow returned an error
            // (the most likely cause for error being that the
            // application is out of memory)

            if ( err != noErr )
            {
                DisposeWindow (*window);
            }

        }

        return err;
    }
```

Additionally, with the Mac OS 8.5 Window Manager, there are two new ways to create a window:

- from an extensible window resource (a resource of type `'wind'`), with the function `CreateWindowFromResource` (page 69)
- from a Collection Manager collection, with the function `CreateWindowFromCollection` (page 68)

A collection is an abstract data type, defined by the Collection Manager, that allows you to store multiple pieces of related information. For purposes of the Window Manager, however, a collection might best be understood as an intermediate state between a live window and a `'wind'` resource. Using the function `StoreWindowIntoCollection` (page 70), your application can store any window, even those not created with Mac OS 8.5 Window Manager functions, into a collection. You can also store data associated with the window into the same collection. This provides a quick way for your application to save a simple document.

From a collection, your application can create a flattened collection—that is, a stream of address-independent data—using the Collection Manager. Because the `'wind'` resource consists of an extensible flattened collection, your application can store a flattened collection consisting of a window and its data into a `'wind'` resource using the Resource Manager. “Storing a Document Window Into a Collection” (page 57) provides an example of how your application might store a window and its data as a single flattened collection in an extended `'wind'` resource.

Enabling Floating Windows

If you wish to enable system support for floating windows, you must initialize the Window Manager by calling the function `InitFloatingWindows` (page 75) before using any other Window Manager functions. Your application calls the `InitFloatingWindows` function—instead of the `InitWindows` function—to initialize the Window Manager and enable automatic front-to-back display ordering of all your application's windows. When your application calls `InitFloatingWindows`, the system automatically sorts each of your application's windows into one of three window display layers: modal, floating, and document. As with `InitWindows`, before calling `InitFloatingWindows`, you must initialize QuickDraw and the Font Manager by calling the `InitGraf` and `InitFonts` functions, respectively.

IMPORTANT

The `InitFloatingWindows` function is supported under Mac OS 8.6 and later. ▲

See "Responding to Suspend and Resume Events" (page 50) for an example of how you can use the Mac OS 8.5 Window Manager to hide and show floating windows when your application receives suspend and resume events.

Positioning a Window on the Desktop

Your goal in positioning a window on the desktop is to place it where the user expects it. For a new document, this usually means just below and to the right of the last document window in which the user was working. For a saved document, it usually means the location of the document window when the document was last saved (if it was saved on a computer with the same screen configuration).

On Macintosh computers with a single screen of known size, positioning windows is fairly straightforward. You position the first new document window on the upper-left corner of the desktop. Open each additional new document window with its upper-left corner slightly below and to the right of the upper-left corner of its predecessor.

On computers with multiple monitors, window placement depends on a number of factors:

- the number of screens available and their dimensions
- the location of the main screen—that is, the screen that contains the menu bar
- the location of the screen on which the user was most recently working

In general, you place the first new document window on the main screen, and you place subsequent document windows on the screen that contains the largest portion of the most recently active document window. That is, if you display a blank document window when the user starts up your application, you place the window on the main screen. If the user moves the window to another screen and then creates another new document, you place the new document window on the other screen. Although the user is free to place windows so that they cross screen boundaries, you should never display a new window that spans multiple screens.

When the user opens a saved document, you replicate the size and location of the window in which the document was last saved, if possible.

The Window Manager recognizes a set of positioning constants—which you supply in the extended window (`'wind'`) resource or via the function `RepositionWindow` (page 94)—that let you position new windows automatically. You typically use the constant `kWindowCascadeOnParentWindowScreen` for positioning document windows. The `kWindowCascadeOnParentWindowScreen` constant specifies the basic guidelines for document window placement: The Window Manager places the first window in the upper-left corner of the main screen. It places subsequent windows with their upper-left corners below and to the right of the upper-left corner of the last window in which the user was working. The exact amount of pixels that the subsequent windows are shifted depends upon the current appearance.

If the user moves or closes a window that occupies one of the interim positions, and the window template specifies `kWindowCascadeOnParentWindowScreen`, the Window Manager uses the “empty” slot for the next new window created before moving further down and to the right.

For a complete list of the positioning constants and their effects, see “RepositionWindow Constants” (page 136).

Supporting Window Proxy Icons

With Mac OS 8.5, document windows support the display of a small icon in the window's title bar, next to the window title, that serves as a proxy for the document's icon in the Finder. This proxy icon should appear and behave the way the document's icon does in the Finder.

Your application can call the function `SetWindowProxyCreatorAndType` (page 102) when you want to establish a proxy icon for a window, but the window's data has not yet been saved to a file. By passing `SetWindowProxyCreatorAndType` the creator and type of the file that the window is to contain, you can provide visual consistency with other windows that have saved files and with the Finder. Listing 2-1 in "Creating a Window" (page 32) provides an example of a simple function for creating and displaying a window that includes using `SetWindowProxyCreatorAndType` to establish a proxy icon.

If the window's data has been saved to a file, your application can call the functions `SetWindowProxyFSSpec` (page 104) or `SetWindowProxyAlias` (page 101) to associate the file with the window and thereby establish the proxy icon.

Once a window has a proxy icon, the user should be able to manipulate it as if they were performing actions with a Finder icon for the window's file. For example, if a user drags a proxy icon to a folder, Finder window, the desktop, or another volume, the file represented by the proxy icon should be moved or copied accordingly, as if the user had dragged the file's icon in the Finder.

Your application detects a proxy icon drag when the function `FindWindow` returns the `inProxyIcon` result code, and it can use Window Manager-supplied functions to handle the drag process. If the proxy icon represents an object type handled by the Window Manager (currently, files), the Window Manager can handle all aspects of the drag process itself, and your application should simply call the function `TrackWindowProxyDrag` (page 112). If your application calls the `TrackWindowProxyDrag` function it does not have to call the Drag Manager function `WaitMouseMoved` before starting to track the drag, as the Window Manager handles this automatically. "Tracking a Window Proxy Icon Drag" (page 45) provides an example of how your application might call `TrackWindowProxyDrag`.

Because a user can so readily use a proxy icon to manipulate a document file while the document is itself open, your application should call a function in its

event loop to synchronize the file data for all of its document windows. While keeping your application's file data synchronized with that of the Finder is a good practice in general, it is especially important if your application is using proxy icons in its document windows. Because a proxy icon is much more prominent to a user than a Finder icon when the user is working in an open document, it is therefore more likely that the user may move the file represented by the proxy icon while the document is open.

For example, if a user opens "My Document" in an application, then drags the proxy icon for "My Document" to a different folder, the application may still expect "My Document" to be in its original location. Additionally, the user may change the name of "My Document" to "Your Document" or place "My Document" in the Trash folder while "My Document" is open.

Optimally, your application should synchronize itself with the actual location of files on disk after every call to `WaitNextEvent`. This is preferable to performing file synchronization after calling `TrackWindowProxyDrag`, because some time may elapse between the time `TrackWindowProxyDrag` returns and the time that the file is actually moved on disk.

The application-defined function `MySynchronizeFiles` shown in Listing 2-2 is intended to be called after every call to `WaitNextEvent`. For each of an application's document windows, `MySynchronizeFiles` updates the application's internal data structures to match that of the file as it exists on disk. The `MySynchronizeFiles` function additionally ensures that the name of the document window is changed to match the name of the file on disk, and closes the document window if the file is moved to the Trash folder.

Listing 2-2 Synchronizing files for all document windows

```
static void MySynchronizeFiles( void )
{

    // File synchronization for all document windows

    static UInt32   nextSynchTicks = 0;
    UInt32          currentTicks;
    WindowPtr       currentWindow;
    OSStatus        trashStatus;
    SInt16          trashVRefNum;
    SInt32          trashDirID;
```

```
currentTicks = TickCount();
currentWindow = FrontNonFloatingWindow();

// Find the Trash folder
trashStatus = FindFolder(   kOnSystemDisk,
                            kTrashFolderType,
                            kDontCreateFolder,
                            &trashVRefNum,
                            &trashDirID );

if( currentTicks > nextSynchTicks )
{
    // Loop over all document windows,
    // searching for files whose locations have changed
    while ( currentWindow != NULL )
    {
        // Note: DocumentWindowData is a placeholder for
        // your application's document data structure
        DocumentWindowData *documentWindowData = GetWRefCon(currentWindow);

        // If the window is owned by this application...
        if( documentWindowData != NULL )
        {
            Boolean     aliasChanged;
            FSSpec      newSpec;
            FolderType  folder;

            aliasChanged = false;
            folder = 0;

            // Ask the Alias Manager for the document's file location...
            (void) ResolveAlias(    NULL,
                                    documentWindowData->fileAlias,
                                    &newSpec,
                                    &aliasChanged );
            if( aliasChanged )
            {
                // The file location has changed; update the window
                documentWindowData->fileSpec = newSpec;
```

```
        // The user might have renamed the file
        SetWTitle( currentWindow, newSpec.name );

    }

    // Close the document if the user moved the file into the Trash
    //
    // We need to walk up the file's parent folder hierarchy to ensure
    // that the user hasn't moved it into a folder inside the Trash
    //
    // We ignore the aliasChanged flag because the parent folder
    // hierarchy can change without affecting the alias

    if( trashStatus == noErr )
    {
        do
        {
            // If we've reached a root folder, we know
            // the file's not in the Trash

            if( newSpec.parID == fsRtParID )
                break;

            // If the Trash is a parent of the original file,
            // close the window

            if( (newSpec.vRefNum == trashVRefNum)
                &&  (newSpec.parID == trashDirID)
                )
            {
                // Your app's "close document window" code goes here...
                break;
            }
        } while( FSMakeFSSpec(  newSpec.vRefNum,
                                newSpec.parID,
                                "\p",
                                &newSpec ) == noErr );

    }
}
```

```
            currentWindow = GetNextWindow( currentWindow );
        }

        // To avoid calling ResolveAlias too often, wait at least
        // 1/4 second between synchronization iterations
        nextSynchTicks = ( currentTicks + 15 );
    }
}
```

Applications typically track the modification state of a document in order to inform the user that they have made changes to the document which they might wish to save before closing the window. Your application should inform the Window Manager when a document has unsaved changes by calling the function `SetWindowModified` (page 111). When you do so, the Window Manager displays the document's proxy icon in a disabled state and prevents the user from dragging the proxy icon. Disabled proxy icons cannot be dragged because unsaved documents cannot be moved or copied in a manner predictable to the user. "Setting a Window's Modification State" (page 57) provides an example of how your application might call `SetWindowModified`.

IMPORTANT

The only time that a document's proxy icon should be disabled is when the document has unsaved changes. Your application should not disable the proxy icon at any other time. ▲

Finally, when the user drags content that your application can accept into the content area of one of its windows, the window's structure region, including the proxy icon, should become highlighted. This gives visual feedback that the window is a valid destination for the content. Applications typically call the Drag Manager functions `ShowDragHilite` and `HideDragHilite` to indicate that a window is a valid drag-and-drop destination. If your application does not do this—that is, if your application implements any type of custom drag highlighting, such as highlighting more than one area of a window at a time—it must call the function `HiliteWindowFrameForDrag` (page 110).

Drawing in a Window's Content Region

Your application and the Window Manager work together to display windows on the screen. Once you have created a window and made it visible, the Window Manager automatically draws the window's structure region (that is, its "frame") in the appropriate location. The Window Manager does not typically draw any content in a window; it only draws the color or pattern of the content region. Your application is responsible for drawing content such as text or graphics in the window's content region.

When the user exposes a window that has previously been obscured, the Window Manager redraws the exposed, invalid portions of the window. If some part of the window's content region is exposed, the Window Manager redraws it to the current content color and adds it to the window's update region.

You can set a window's content color by calling the function `SetWindowContentColor` (page 84). As shown in Listing 2-3, when your application calls `SetWindowContentColor`, the Window Manager uses this content color to redraw the content region of the window. Your application can use the function `SetWindowContentPattern` (page 85) to specify a pattern for the content region.

Listing 2-3 Setting the window's content color to red

```
static OSStatus MySetWindowContentColorToRed( WindowPtr window )
{
    RGBColor    rgbRedColor = { 0xFFFF, 0, 0 };  // red, green, blue
    OSStatus    outStatus;

    outStatus = SetWindowContentColor( window, &rgbRedColor );
    return outStatus;
}
```

None of the Mac OS 8.5 functions affect the window's graphics port's background color or pattern. However, `SetWindowContentColor` and `SetWindowContentPattern` do supersede the window color table structure, the

'wctb' resource, and the SetWinColor function, none of which are supported under Carbon.

Handling Window Events

Your application must be prepared to handle two basic kinds of window-related events in its event loop:

- mouse and keyboard events, which are reported by the Event Manager in direct response to user actions
- activate, update, suspend, and resume events, which are generated by the Window Manager and the Event Manager as an indirect result of user actions

Your application receives mouse-down events if it is the foreground process and the user clicks in the menu bar or a window belonging to your application. When it receives a mouse-down event, your application first calls the FindWindow function to map the cursor location to a window region, and then it branches to one of its own functions. See “Responding to Mouse-Down Events” (page 44) for a further discussion of handling mouse-down events under Mac OS 8.5.

Whenever your application is the foreground process, it receives keyboard events. When the user presses a key or a combination of keys, your application responds by inserting data into the document, changing the display, or taking other actions as defined by your application.

Your application activates and deactivates windows in response to activate events, which the Event Manager generates to inform your application that a window is becoming active or inactive. Each activate event specifies the window to be changed and the direction of the change (that is, whether it is to be activated or deactivated).

The Event Manager sends your application an update event when changes on the desktop or in a window require that part or all of a window’s content region be updated. The Window Manager and your application can both trigger update events by adding regions that need updating to the update region, as described in the section “Maintaining the Update Region” (page 52).

A switch into or out of your application from a different application is handled through suspend and resume events, not activate events. For example, if the user clicks in a window belonging to another application, the Event Manager

typically sends your application a suspend event and performs a major switch to the other application. One of the ways that your application handles a suspend or resume event is by hiding or showing its floating windows; see “Responding to Suspend and Resume Events” (page 50) for details.

In addition to handling specific events, however, your application should also call a function in its event loop to synchronize the file data for all of its document windows. While keeping your application’s file data synchronized with that of the Finder is a good practice in general, it is especially important if your application is using proxy icons in its document windows. Because a proxy icon is much more prominent to a user than a Finder icon when the user is working in an open document, it is therefore more likely that the user may move the file represented by the proxy icon while the document is open. See “Supporting Window Proxy Icons” (page 37) for a sample file synchronization function and a description of other aspects of proxy icon management.

Responding to Mouse-Down Events

When your application receives a mouse-down event, your application calls the `FindWindow` function to map the cursor location to a window region. The `FindWindow` function specifies the region by returning one of these constants:

Constant	Description
`inDesk` or `inNoWindow`	None of the following
`inMenuBar`	The menu bar
`inSysWindow`	A desk accessory window
`inContent`	Anywhere in the content region except the size box if the window is active; anywhere including the size box if the window is inactive
`inDrag`	The drag region (in a window that contains a proxy icon, the drag region excludes the proxy icon region)
`inGrow`	The size box (of an active window only)
`inGoAway`	The close box
`inZoomIn`	The zoom box (when the window is in the standard state)

Constant	Description
inZoomOut	The zoom box (when the window is in the user state)
inCollapseBox	The collapse box
inProxyIcon	The proxy icon

When the user presses the mouse button while the cursor is in a window, FindWindow not only returns a constant that identifies the window region but also reports which window the cursor is in by placing a pointer to this window at the address specified in one of its parameters. Your response to FindWindow depends on whether the cursor is in the active window and the kind of window that the cursor is in.

When you receive a mouse-down event in the active window, you route the event to the function that is appropriate for handling the mouse-down event for a given region. "Tracking a Window Proxy Icon Drag" (page 45) describes how your application can respond to a mouse-down event in the proxy icon region of a window that indicates a user request to drag the proxy icon. "Displaying a Window Path Pop-Up Menu" (page 46) shows how your application can handle the case where a mouse-down event—in either a window's drag region or its proxy icon region—indicates a user request to display the window path pop-up menu.

Tracking a Window Proxy Icon Drag

A mouse-down event in the proxy icon region of a document window can indicate that the user either wishes to drag the proxy icon or wishes to display the path pop-up menu for the window. Listing 2-5 in "Displaying a Window Path Pop-Up Menu" (page 46) provides an example of how your application can respond to receiving the inProxyIcon result from the FindWindow function if the user is not dragging the proxy icon.

If the user is dragging the proxy icon, your application can use Window Manager-supplied functions to handle the drag process. If the proxy icon represents a type of object (currently, file system entities such as files, folders, and volumes) that the Window Manager supports, the Window Manager can handle all aspects of the drag process itself, and your application should simply call the function TrackWindowProxyDrag (page 112), as shown in Listing 2-4. If your application calls the TrackWindowProxyDrag function it does not have to call the Drag Manager function WaitMouseMoved before starting to track the drag, as the Window Manager handles this automatically.

If the proxy icon represents an object type other than a file (other object types are currently not handled by the Window Manager), or if you wish to implement custom dragging behavior, your application should call the function `TrackWindowProxyFromExistingDrag` (page 113). The `TrackWindowProxyFromExistingDrag` function accepts an existing drag reference and adds file data if the window contains a file proxy.

If your application uses `TrackWindowProxyFromExistingDrag`, you have the choice of using this function in conjunction with the functions `BeginWindowProxyDrag` (page 107) and `EndWindowProxyDrag` (page 108) or simply calling `TrackWindowProxyFromExistingDrag` and handling all aspects of creating and disposing of the drag yourself.

Listing 2-4 Tracking a window proxy icon drag within the event loop

```
case inProxyIcon:

// We've seen a hit in the window proxy area, so drag the proxy icon

// Note that we don't check that the
// window is an app window, but you should

{
    OSStatus status = TrackWindowProxyDrag( pWindow, pEvent->where );
    if( status == errUserWantsToDragWindow )
        handled = false;
    else if( status == noErr )
        handled = true;
}
// Fall through to checking whether the user
// wants to display a window path pop-up menu
```

Displaying a Window Path Pop-Up Menu

The Mac OS 8.5 Window Manager provides system support for your application to display window path pop-up menus—like those used in Finder windows. When the user presses the Command key and clicks on the window's title, the window displays a pop-up menu containing a standard file system path, informing the user of the location of the document displayed in the window and allowing the user to open windows for folders along the path.

Because the window title includes both the proxy icon region and part of the drag region of the window, your application must be prepared to respond to a click in either region by displaying a window path pop-up menu. Therefore, when the `FindWindow` function returns either the `inDrag` or the `inProxyIcon` result code—you should pass the event to the function `IsWindowPathSelectClick` (page 115) to determine whether the mouse-down event should activate the window path pop-up menu. If `IsWindowPathSelectClick` returns a value of `true`, your application should then call the function `WindowPathSelect` (page 116) to display the menu. Listing 2-5 shows how your application might handle a user request to display the window path pop-up menu.

Listing 2-5 Determining whether to display the window path pop-up menu

```
case mouseDown:
    {
    short part = FindWindow(pEvent->where, &pWindow);

    switch ( part )
        {
        case inProxyIcon:

        // We've seen a hit in the window proxy area, so drag the proxy icon
        // Note that we don't check that the window is an app window, but you should
            {
                OSStatus status = TrackWindowProxyDrag( pWindow, pEvent->where );
                if( status == errUserWantsToDragWindow )
                    handled = false;
                else if( status == noErr )

                    handled = true;

            }

        // fall through

        case inDrag:
            if( !handled )
            // Check that we should show the window file path pop-up menu
                {
                // Note that we don't check that the window
```

```
            // is an app window, but you should
                {
                if( IsWindowPathSelectClick( pWindow, pEvent ) )
                    {
                    SInt32 itemSelected;

                    if(WindowPathSelect( pWindow, NULL, &itemSelected ) == noErr )
                        {
                        // If the menu item selected is not the title of the window
                        // itself, switch to the Finder, since the window chosen
                        // probably isn't visible
                        if( LoWord(itemSelected) > 1 )
                            {
                            MyBringFinderToFront();
                            }
                        }

                    handled = true;
                    }
                }

            if( !handled )
                {
                // Call DragWindow and drag the window
                ...
                }
            }
        break;
    }
break;
}
```

Note that in Listing 2-5, the user may have selected a menu item for a folder representing a Finder window from the window path pop-up menu. Your application must ensure that the resulting window is visible to the user by making the Finder the frontmost process, as is shown in Listing 2-6.

Listing 2-6 Bringing the Finder to the front

```
static void MyBringFinderToFront(void)
{
    const OSType        kFinderSignature = 'MACS';
    const OSType        kFinderType = 'FNDR';
    ProcessSerialNumber finderProcess;

    // If we find the Finder...
    if( MyFindProcess( kFinderSignature, kFinderType, &finderProcess ) == noErr)
    {
        // Tell the Process Manager to bring the Finder to the front
        (void) SetFrontProcess( &finderProcess );
    }
    else
    {
        // If the Finder can't be brought up, alert the user
        ...
    }
}
```

As shown in Listing 2-6, making the Finder the frontmost process requires that your application call the Process Manager function `SetFrontProcess` with the Finder's process serial number to bring the Finder to the front. Listing 2-7 provides an example of how your application may obtain the Finder's process serial number.

Listing 2-7 Finding the process serial number of a process

```
// Find the PSN of a process, in this case, the Finder,
// given a type and creator pair corresponding to the type
// and creator of the file from which the process was launched.

static OSStatus MyFindProcess( OSType creator, OSType type, ProcessSerialNumber
*outProcess )
{
    ProcessInfoRec      theProc;
    OSStatus            outStatus = 0L;
    ProcessSerialNumber psn;
```

```
    // Start from kNoProcess
    psn.highLongOfPSN = 0;
    psn.lowLongOfPSN = kNoProcess;

    // Initialize ProcessInfoRec fields, or we'll have memory hits in random locations
    theProc.processInfoLength = sizeof( ProcessInfoRec );
    theProc.processName = nil;
    theProc.processAppSpec = nil;
    theProc.processLocation = nil;

    while(true)
    {
        // Keep looking for the process until we find it
        outStatus = GetNextProcess(&psn);
        if( outStatus != noErr )
            break;

        // Is the current process the one we're looking for?
        outStatus = GetProcessInformation(&psn, &theProc);
        if( outStatus != noErr )
            break;
        if( (theProc.processType == type ) && (theProc.processSignature == creator) )
            break;
    }
    *outProcess = psn;
    return outStatus;
}
```

Responding to Suspend and Resume Events

The Event Manager function `WaitNextEvent` returns a suspend event when your application is about to be switched to the background. `WaitNextEvent` returns a resume event when your application becomes the foreground process again.

Upon receiving a suspend event, your application should deactivate the front window and hide any floating windows. Upon receiving a resume event, your application should activate the front window and restore any windows to the state the user left them in at the time of the previous suspend event. For example, your application should show scroll bars and any floating windows. Listing 2-8 provides an example of how your application can respond to a

suspend or resume event by calling the Window Manager functions `HideFloatingWindows` (page 74) and `ShowFloatingWindows` (page 77) to hide or show its floating windows, respectively.

IMPORTANT

The `HideFloatingWindows` and `ShowFloatingWindows` functions are supported under Mac OS 8.6 and later. ▲

Listing 2-8 Hiding and showing floating windows

```
case suspendResumeMessage:
{
    // The message field of the EventRecord indicates whether you are
    // activating (resumeFlag is 1) or deactivating (resumeFlag is 0)

    Boolean becomingActive = (pEvent->message & resumeFlag) != 0;

    // The first document window should be activated or deactivated
    // in response to a suspendResumeMessage, since no other explicit
    // activate message will be sent to your application

    WindowPtr pWindow = FrontNonFloatingWindow();
    if (pWindow != NULL)
        MyHandleActivateDeactivateEvent(pWindow, becomingActive);

    // Human interface standards specify that floating windows be
    // shown when your application becomes active, and hidden while
    // it is inactive

    if (becomingActive)
        (void) ShowFloatingWindows();
    else
        (void) HideFloatingWindows();
    break;
}
```

Maintaining the Update Region

The Window Manager helps your application keep the window display current by maintaining an update region, which represents the parts of your content region that have been affected by changes to the desktop. If a user exposes part of an inactive window by dragging an active window to a new location, for example, the Window Manager adds the newly exposed area of the inactive window to that window's update region.

When your application calls the Event Manager function `WaitNextEvent` and there are no events queued, the Event Manager scans the update regions of all windows on the desktop. If it finds one whose update region is not empty, it generates an update event for that window. When your application receives an update event, it redraws as much of the content area as necessary. Note that your application can receive update events when it is in either the foreground or the background.

Your application can force and suppress update events by manipulating the update region using Window Manager functions provided for this purpose.

You can remove an area from the update region by calling either of the functions `ValidWindowRect` (page 124) or `ValidWindowRgn` (page 125), when you know that the area is in fact valid. Limiting the size of the update region decreases the time that your application spends redrawing window content in response to update events.

The functions `ValidWindowRect` and `ValidWindowRgn` each inform the Window Manager that an area of a window no longer needs to be redrawn. The functions are, respectively, similar to the earlier Window Manager functions `ValidRect` and `ValidRgn`, but the Mac OS 8.5 functions allow the window that they operate upon to be explicitly specified, instead of operating on the current graphics port, so they do not require the graphics port to be set before their use.

You can add an area to the update region by calling either of the functions `InvalWindowRect` (page 122) or `InvalWindowRgn` (page 123). Each function informs the Window Manager that an area of a window should be redrawn.

Moving a Window

When the user drags a window, the window should move, following the cursor as it moves on the desktop. By calling the pre–Mac OS 8.5 Window Manager function `DragWindow`, your application lets the user move the window. When your application wishes to move a window for a reason other than a user-instigated drag, however, it should use either the Mac OS 8.5 Window Manager function `MoveWindowStructure` (page 93) or the earlier function `MoveWindow`.

On versions of the Mac OS that include the Appearance Manager, the size and shape of a window frame may vary from appearance to appearance. Because of this, the total dimensions of a window (that is, the window's structure region) may also vary, causing the window's spatial relationship to the rest of the screen to change. Your application can best accommodate variable window dimensions by using Window Manager functions to size and position your windows, rather than via constant dimensions. Using these functions also allows your application to avoid maintaining its own table of window definition IDs and their various border dimensions, as well as ensuring your application's support of future window definitions.

The `MoveWindowStructure` function moves the specified window, but does not change the window's size. When your application calls `MoveWindowStructure`, the positioning of the specified window is determined by the positioning of its structure region. This is in contrast to the `MoveWindow` function, where the positioning of the window's content region determines the positioning of the window.

The function `SetWindowBounds` (page 96) also provides a means of moving a window, but you would typically call `SetWindowBounds` when you wish to set the size of a window as well. The `SetWindowBounds` function sets a window to the size and position of a rectangle that you specify, and it can interpret this rectangle as the bounding rectangle for either the window's structure or content region (your choice).

In general, you should specify the structure region as the determining basis if how the window as a whole relates to a given monitor is more important than the exact positioning of its content on the screen. If you specify the content region—because the positioning of your application's content is of greatest concern—it is important to note that under some appearances some part of the

window's structure region or "frame" may extend past the edge of a monitor and not be displayed.

Finally, setting a window's position may also be done algorithmically, via the function `RepositionWindow` (page 94), which positions a window relative to another window or a display screen. See "Positioning a Window on the Desktop" (page 35) for a discussion of algorithmic window positioning on Mac OS 8.5.

Zooming a Window Gracefully

When the user clicks a window's zoom box, a window zooms between two states, the user state and the standard state. The user state is any size and position in which the user can place the window on the desktop. The standard state is the size and position that the application defines as being best for the display of the data contained in the window. There are human interface guidelines, described in "Window Zooming" (page 21), that describe how best to determine a window's standard state, based upon its current user state, but prior to Mac OS 8.5 there were no system-supplied functions that enforced these guidelines for your application.

With Mac OS 8.5, you can use the Window Manager function `ZoomWindowIdeal` (page 90) instead of the older function `ZoomWindow` to zoom a window. When your application calls `ZoomWindowIdeal`, it automatically conforms to the human interface guidelines for determining a window's standard state. Using `ZoomWindowIdeal` in conjunction with the Mac OS 8.5 Window Manager functions `SetWindowIdealUserState` (page 89) and `GetWindowIdealUserState` (page 86) also ensures that the user state is tracked accurately, as well as giving your application access to a window's user state in a Carbon-compliant manner.

Note that if your application uses `ZoomWindowIdeal`, the `WStateData` structure is superseded, and the result of the `FindWindow` function should be ignored when determining whether a particular user click on the zoom box is a request to zoom in or out. When you adopt `ZoomWindowIdeal` and your application receives a result of either `inZoomIn` or `inZoomOut` from `FindWindow`, your application should use the function `IsWindowInStandardState` (page 87) and code such as that in Listing 2-9 to determine the appropriate part code to pass into the `partCode` parameter.

Listing 2-9 Determining the appropriate part code to supply to ZoomWindowIdeal

```
switch (FindWindow(myEvent.where, &window))
{

// If FindWindow returns a part code for the zoom box, don't rely on it;
// call IsWindowInStandardState with your application-defined ideal
// window size to figure out whether the window is currently zoomed in or
// out and, therefore, what the part code should be

    case inZoomIn:
    case inZoomOut:
    {
        int     part;
        Point   idealSize = MyFigureWindowIdealSize(window);

        // If IsWindowInStandardState returns true, the window is
        // currently zoomed out to the standard state, so the mouse-down
        // event in the zoom box should be interpreted as inZoomIn

        if (IsWindowInStandardState(window, &idealSize, NULL))
        {
            part = inZoomIn;
        }

        else
        {

        // If IsWindowInStandardState returns false, the window is
        // currently zoomed in to the user state, so the mouse-down event
        // in the zoom box should be interpreted as inZoomOut

            part = inZoomOut;
        }

        // If TrackBox confirms that the mouse-up event occurred while
        // the cursor was still over the zoom box, give ZoomWindowIdeal
        // the real part code, so it can get on with zooming

        if (TrackBox(window, myEvent.where, part))
        {
```

```
            ZoomWindowIdeal(window, part, &idealSize);
        }
        break;
    }
}
```

Resizing a Window

The size box, in the lower-right corner of a window's content region, allows the user to change a window's size. When the user positions the cursor in the size box and presses the mouse button, your application can call the Window Manager's `ResizeWindow` (page 95) function. This function displays a grow image—an outline of the window's frame and scroll bar areas, which expands or contracts as the user drags the size box. The grow image indicates where the window edges would be if the user released the mouse button at any given moment.

The `ResizeWindow` function moves the grow image around the screen, following the user's cursor movements, and handles all user interaction until the mouse button is released. Unlike with the function `GrowWindow`, there is no need to follow this call with a call to the function `SizeWindow`, because once the mouse button is released, `ResizeWindow` resizes the window if the user has changed the window size. Once the resizing is complete, `ResizeWindow` draws the window in the new size.

Your application should call the `ResizeWindow` function instead of the earlier Window Manager functions `SizeWindow` and `GrowWindow`. Some appearances may allow the window to be resized from any corner, not just the bottom right, and as a result, when the user resizes the window, the window may move on the screen and not merely change size. `ResizeWindow` informs your application of the new window bounds, so that your application can respond to any changes in the window's position.

IMPORTANT

The `ResizeWindow` function is supported under Mac OS 8.6 and later. ▲

To avoid an unmanageably large or small window, you supply the lower and upper size limits for the window in the `sizeConstraints` parameter of

`ResizeWindow`. Note that although you supply `ResizeWindow` with the size limits via a structure of type `Rect`, the values referenced through the `sizeConstraints` parameter represent window dimensions, not screen coordinates.

Setting a Window's Modification State

Your application should inform the Window Manager when a document has unsaved changes by calling the function `SetWindowModified` (page 111). When you do so, the Window Manager displays the document's proxy icon in a disabled state and prevents the user from dragging the proxy icon. Disabled proxy icons cannot be dragged because unsaved documents cannot be moved or copied in a manner predictable to the user. Listing 2-10 provides an example of how your application might call `SetWindowModified` to set the modified state of a window.

IMPORTANT

The only time that a document's proxy icon should be disabled is when the document has unsaved changes. Your application should not disable the proxy icon at any other time. ▲

Listing 2-10 Setting the modified state for a window

```
void MySetDocumentContentChanged( WindowDataPtr pData, Boolean changed )
{
    pData->changed = changed;
    SetWindowModified( (WindowPtr)pData, changed );
}
```

Storing a Document Window Into a Collection

Using the function `StoreWindowIntoCollection` (page 70), your application can store any window, not just those created with Mac OS 8.5 Window Manager functions, into a collection. You can also store data associated with the window

into the same collection. This provides a quick way for your application to save a simple document.

From a collection, your application can create a flattened collection—that is, a stream of address-independent data—using the Collection Manager. Because the `'wind'` resource consists of an extensible flattened collection, your application can store a flattened collection consisting of a window and its data into a `'wind'` resource using the Resource Manager. Listing 2-11 provides an example of how your application might store a window and its data as a single flattened collection in an extended `'wind'` resource.

Listing 2-11 Writing a document window into a flattened collection resource

```
enum
{
    kDocumentResType    = 'Docu', // 'Docu' is an extended 'wind' resource
    kResID_Document     = 128
};

static pascal OSStatus MyWriteDocumentFile (WindowPtr window, short fileRefNum)
{
    OSStatus err = noErr;

    TEHandle teHandle = (TEHandle) GetWRefCon (window);

        Collection  collection  = nil;
        Handle      flatDoc     = nil;
        Handle      flatDocRes  = nil;

        do
        {
            // Temporarily create a collection into which the Window Manager will put
            // a description of the window

            if (!(collection = NewCollection ( )))
            {
                err = MemError ( );
                break;
            }
```

```
// Store the window into the collection

err = StoreWindowIntoCollection (window, collection);
if (err != noErr) break;

// Stash a copy of the text into the collection

err = AddCollectionItemHdl (collection, 'TEXT', 1, (**teHandle).hText);
if (err != noErr) break;

// Allocate a new 0-length handle to hold the flattened collection

flatDoc = NewHandle (0);
if (!flatDoc)
{
    err = MemError ( );
    break;
}

// Flatten the collection into the handle

err = FlattenCollectionToHdl (collection, flatDoc);
if (err != noErr) break;

// Save the flattened collection as a resource in the file
// whose resource map is topmost in the chain

AddResource (flatDoc, kDocumentResType, kResID_Document, "\p");
err = ResError ( );
if (err != noErr) break;

flatDocRes = flatDoc;
flatDoc = nil;

WriteResource (flatDocRes);
err = ResError ( );
if (err != noErr) break;

// We've changed the resource map, so force it to be updated on disk

UpdateResFile (fileRefNum);
```

```
            err = ResError ( );
            if (err != noErr) break;

            // The document has been written, so it's OK to say so

            err = SetWindowModified (window, false);
            if (err != noErr) break;
        }
        while (false);

        if (collection)
            DisposeCollection (collection);
        if (flatDocRes)
            ReleaseResource (flatDocRes);
        if (flatDoc)
            DisposeHandle (flatDoc);

    return err;
}
```

Mac OS 8.5 Window Manager Reference

Contents

This chapter describes the Window Manager application programming interface (API) introduced with Mac OS 8.5 and Appearance Manager 1.1, as follows:

Note that the preexisting Window Manager API is not discussed in this document. For a description of the Mac OS 8 Window Manager API, see *Mac OS 8 Window Manager Reference*. For descriptions of the pre–Mac OS 8 Window Manager API, see *Inside Macintosh: Macintosh Toolbox Essentials*.

Gestalt Constants

Before calling any functions dependent on the Window Manager, your application should pass the selector `gestaltWindowMgrAttr` to the `Gestalt` function to determine which Window Manager functions are available.

```
enum {
    gestaltWindowMgrAttr        = 'wind',
    gestaltWindowMgrPresent     = (1L << 0),
    gestaltHasFloatingWindows   = 2
};
```

Constant descriptions

`gestaltWindowMgrAttr`
: The `Gestalt` selector passed to determine what features of the Window Manager are present. This selector is available with Mac OS 8.5 and later. The `Gestalt` function produces a 32-bit value whose bits you should test to determine which Window Manager features are available.

`gestaltWindowMgrPresent`
: If the bit specified by this mask is set, the Window Manager functionality for Appearance Manager 1.1 is available. This bit is set for Mac OS 8.5 and later.

`gestaltWindowMgrPresent`
: If this bit is set, the functions `InitFloatingWindows` (page 75), `HideFloatingWindows` (page 74), `ShowFloatingWindows` (page 77), and `AreFloatingWindowsVisible` (page 73) are supported. This bit is set for Mac OS 8.6 and later.

Functions

The Mac OS 8.5 Window Manager provides new functions in the following areas:

- "Creating and Storing Windows" (page 67)
- "Referencing Windows" (page 71)
- "Displaying Floating Windows and Window Animations" (page 72)
- "Accessing Window Information" (page 79)
- "Manipulating Window Color Information" (page 82)
- "Zooming Windows" (page 86)
- "Sizing and Positioning Windows" (page 91)
- "Establishing Proxy Icons" (page 97)
- "Coordinating Proxy Icons With Drag-and-Drop Management" (page 106)
- "Activating Window Path Pop-Up Menus" (page 115)
- "Associating Data With Windows" (page 118)
- "Maintaining the Update Region" (page 122)

Creating and Storing Windows

The Mac OS 8.5 Window Manager provides the following functions to create and store windows:

- `CreateNewWindow` (page 67) creates a window from parameter data.
- `CreateWindowFromResource` (page 69) creates a window from `'wind'` resource data.
- `CreateWindowFromCollection` (page 68) creates a window from collection data.
- `StoreWindowIntoCollection` (page 70) stores data describing a window into a collection.

CreateNewWindow

Creates a window from parameter data.

```
pascal OSStatus CreateNewWindow (
                    WindowClass windowClass,
                    WindowAttributes attributes,
                    const Rect *bounds,
                    WindowPtr *outWindow);
```

`windowClass` A value of type `WindowClass`. You pass a `WindowClass` constant that categorizes the type of window to be created. The window class cannot be altered once the window has been created. See "Window Class Constants" (page 140) for a description of possible values for this parameter.

`attributes` An unsigned 32-bit value of type `WindowAttributes`. You set the bits in a `WindowAttributes` field to specify certain features and logical attributes of the window to be created. See "Window Attribute Constants" (page 138) for descriptions of possible values for this parameter.

`bounds` A pointer to a structure of type `Rect`. Before calling `CreateNewWindow`, set the rectangle to specify the size and position of the new window's content region, in global coordinates.

`outWindow` A pointer to a value of type `WindowPtr`. On return, the window pointer points to the newly created window.

function result A result code. See “Result Codes” (page 148).

DISCUSSION

The `CreateNewWindow` function creates a window based on the attributes and class you specify in the `attributes` and `windowClass` parameters. `CreateNewWindow` sets the new window’s content region to the size and location specified by the rectangle passed in the `bounds` parameter, which in turn determines the dimensions of the entire window. The Window Manager creates the window invisibly and places it at the front of the window list. After calling `CreateNewWindow`, you should set any desired associated data—using Window Manager or Control Manager accessor functions—then call the function `TransitionWindow` (page 78) to display the window. See “Creating a Window” (page 32) for a sample application-defined window-creation function.

VERSION NOTES

Available with Mac OS 8.5 and later.

CreateWindowFromCollection

Creates a window from collection data.

```
pascal OSStatus CreateWindowFromCollection (
                    Collection collection,
                    WindowPtr *outWindow);
```

`collection` A reference to the collection to be used in creating the window. You pass a reference to a previously created collection, such as that returned by the Collection Manager function `NewCollection`. The collection used to create the window must contain the required items for a resource of type `'wind'` (page 130) or window creation fails.

`outWindow` A pointer to a value of type `WindowPtr`. On return, the window pointer points to the newly created window.

function result A result code. See “Result Codes” (page 148).

DISCUSSION

The `CreateWindowFromCollection` function creates a window invisibly and places it at the front of the window list. After calling `CreateWindowFromCollection`, you should set any desired associated data—using Window Manager or Control Manager accessor functions—then call the function `TransitionWindow` (page 78) to display the window. The number of references to the collection (that is, its owner count) is incremented by a minimum of one for the duration of this call.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The chapter "Collection Manager" in *Inside Macintosh: QuickDraw GX Environment and Utilities.*

CreateWindowFromResource

Creates a window from `'wind'` resource data.

```
pascal OSStatus CreateWindowFromResource (
                    SInt16 resID,
                    WindowPtr *outWindow);
```

`resID` The resource ID of a resource of type `'wind'` (page 130). Pass in the ID of the `'wind'` resource to be used to create the window.

`outWindow` A pointer to a value of type `WindowPtr`. On return, the window pointer points to the newly created window.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `CreateWindowFromResource` function loads a window from a `'wind'` resource. The Window Manager creates the window invisibly and places it at the front of the window list. After calling `CreateWindowFromResource`, you should set any desired associated data—using Window Manager or Control Manager accessor

functions—then call the function `TransitionWindow` (page 78) to display the window.

VERSION NOTES

Available with Mac OS 8.5 and later.

StoreWindowIntoCollection

Stores data describing a window into a collection.

```
pascal OSStatus StoreWindowIntoCollection (
                    WindowPtr window,
                    Collection collection);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window to be stored.

`collection` A reference to the collection into which the window is to be stored. You pass a reference to a previously created collection, such as that returned by the Collection Manager function `NewCollection`.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `StoreWindowIntoCollection` function stores any window—including those not created by Mac OS 8.5 Window Manager calls—into the specified collection. The Window Manager does not empty the collection beforehand, so any existing items in the collection remain. See "Storing a Document Window Into a Collection" (page 57) for an example of how your application can call `StoreWindowIntoCollection`.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The chapter "Collection Manager" in *Inside Macintosh: QuickDraw GX Environment and Utilities.*

Referencing Windows

The Mac OS 8.5 Window Manager provides the following functions for working with references to windows:

- `GetWindowOwnerCount` (page 72) obtains the number of existing references to a window.
- `CloneWindow` (page 71) increments the number of references to a window.

CloneWindow

Increments the number of references to a window.

```
pascal OSStatus CloneWindow (WindowPtr window);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window whose reference count is to be incremented.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

You should call `CloneWindow` if you are using a window and wish to ensure that it is not disposed while you are using it. With the Mac OS 8.5 Window Manager, all windows are created with a reference count (owner count) of one. The function `CloneWindow` increments the number of references to a window, and the earlier function `DisposeWindow` decrements the number of references. When the reference count reaches zero, `DisposeWindow` disposes of the window.

SPECIAL CONSIDERATIONS

To maintain an accurate reference count, you must follow every call to the `CloneWindow` function with a matching call to the `DisposeWindow` function when your application is ready to release its reference to the window.

VERSION NOTES

Available with Mac OS 8.5 and later.

GetWindowOwnerCount

Obtains the number of existing references to a window.

```
pascal OSStatus GetWindowOwnerCount (
                    WindowPtr window,
                    UInt32 *outCount);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window whose reference (owner) count is to be determined.
`outCount`	A pointer to a value that, on return, contains the current number of references to the window.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

With the Mac OS 8.5 Window Manager, all windows are created with a reference count (owner count) of one. The function `CloneWindow` (page 71) increments the number of references to a window, and the earlier function `DisposeWindow` decrements the number of references. When the reference count reaches zero, `DisposeWindow` disposes of the window.

VERSION NOTES

Available with Mac OS 8.5 and later.

Displaying Floating Windows and Window Animations

The Mac OS 8.5 Window Manager provides the following functions for displaying floating windows:

- `InitFloatingWindows` (page 75) initializes the Window Manager and sorts your application's windows into the proper front-to-back display order.
- `ShowFloatingWindows` (page 77) shows an application's floating windows.

- `HideFloatingWindows` (page 74) hides an application's floating windows.
- `AreFloatingWindowsVisible` (page 73) indicates whether an application's floating windows are currently visible.

The Mac OS 8.5 Window Manager provides the following function for displaying a window with animation and sound:

- `TransitionWindow` (page 78) displays an animation and plays the theme-appropriate sound for a window when it is shown or hidden.

AreFloatingWindowsVisible

Indicates whether an application's floating windows are currently visible.

```
pascal Boolean AreFloatingWindowsVisible (void);
```

function result A value of type `Boolean`. The `AreFloatingWindowsVisible` function returns `true` if the application's floating windows are currently shown. Otherwise, if the application's floating windows are currently hidden, or if the function `InitFloatingWindows` (page 75) has not been called prior to a call to `AreFloatingWindowsVisible`, it returns `false`.

DISCUSSION

When your application receives a suspend event, it must hide any visible floating windows. When your application receives a resume event, it must make its floating windows visible again. If your application needs to check which visibility state its floating windows are in, it may call the `AreFloatingWindowsVisible` function.

SPECIAL CONSIDERATIONS

You should call the function `InitFloatingWindows` (page 75) prior to calling `AreFloatingWindowsVisible`.

The `AreFloatingWindowsVisible` function operates only upon windows created with the `kFloatingWindowClass` constant; see "Window Class Constants" (page 140) for more details on this constant.

VERSION NOTES

Supported with Mac OS 8.6 and later.

HideFloatingWindows

Hides an application's floating windows.

```
pascal OSStatus HideFloatingWindows (void);
```

function result A result code; see "Result Codes" (page 148). Returns `errFloatingWindowsNotInitialized` (-5609) if you have not called `InitFloatingWindows` prior to `HideFloatingWindows`; otherwise, returns `noErr` (0).

DISCUSSION

When your application receives a suspend event, it must hide any visible floating windows. When your application receives a resume event, it must make its floating windows visible again. See "Responding to Suspend and Resume Events" (page 50) for an example of how your application can call the `HideFloatingWindows` function.

SPECIAL CONSIDERATIONS

You must call the function `InitFloatingWindows` (page 75) prior to calling `HideFloatingWindows`.

The `HideFloatingWindows` function operates only upon windows created with the `kFloatingWindowClass` constant; see "Window Class Constants" (page 140) for more details on this constant.

VERSION NOTES

Supported with Mac OS 8.6 and later.

SEE ALSO

The function `ShowFloatingWindows` (page 77).

InitFloatingWindows

Initializes the Window Manager and sorts your application's windows into the proper front-to-back display order.

```
pascal OSStatus InitFloatingWindows (void);
```

function result A result code; see "Result Codes" (page 148). Returns `errWindowsAlreadyInitialized` (-5608) if you have already called either `InitFloatingWindows` or `InitWindows`; otherwise, returns `noErr` (0).

DISCUSSION

Your application calls the `InitFloatingWindows` function—instead of the `InitWindows` function—to initialize the Window Manager and enable automatic front-to-back display ordering of all your application's windows.

Windows can be placed on the display screen so that one window appears to be behind another. This visual overlapping gives the user an impression of depth. A floating window is so-named because its front-to-back display order (that is, its z-order placement relative to other windows on the screen) makes it appear to float in front of document windows.

Because earlier versions of the Window Manager only defined the look of floating windows, not their floating behavior, some applications contain code that implements a floating effect for tool palettes and other such windows. However, your application can now use the Window Manager to automatically sort floating and non-floating windows into separately z-ordered groups, thereby enforcing the proper front-to-back display order.

The Window Manager only enforces display ordering for windows belonging to applications that have explicitly requested this functionality by calling the `InitFloatingWindows` function. Therefore, if you wish to make use of the system-supplied "floating" behavior, you must call `InitFloatingWindows` to initialize the Window Manager, not `InitWindows`. If you use `InitWindows` to initialize the Window Manager, floating windows intermingle with non-floating windows, and your application is still responsible for ensuring that floating windows remain higher in z-order than non-floating windows. However, if you use `InitFloatingWindows`, the Window Manager automatically ensures that floating windows remain higher in z-order than any non-floating window. When your application calls `InitFloatingWindows`, the Window Manager sorts

each of your application's windows into one of three window display layers: modal, floating, and document.

For windows created with the Mac OS 8.5 function `CreateNewWindow` (page 67), the order in which the Window Manager sorts the windows is based on window class. See "Window Class Constants" (page 140) for a description of the various window classes (for a floating window, `kFloatingWindowClass`) which determine sort order for windows created with `CreateNewWindow`.

For pre–Mac OS 8.5 windows, the sort order is based upon window definition ID. For example, if your application calls `InitFloatingWindows`, then calls the `GetNewWindow` function with a `dboxProc` window ID, this produces a dialog box located in the modal display layer. A similar ordering is imposed for floating and document window definition function IDs.

To obtain system support for floating windows, before using any other Window Manager functions you must initialize the Window Manager by calling the `InitFloatingWindows` function. As with `InitWindows`, before calling `InitFloatingWindows`, you must initialize QuickDraw and the Font Manager by calling the `InitGraf` and `InitFonts` functions, respectively.

Also, before calling the `InitFloatingWindows` function you should always confirm that `InitFloatingWindows` is supported by the version of the Mac OS upon which your application is running. To do this, check the value returned by the `Gestalt` function to ensure that the `gestaltHasFloatingWindows` bit is set, as described in "Gestalt Constants" (page 65).

As part of initialization, the `InitFloatingWindows` function creates the Window Manager port, a graphics port that occupies all of the main screen. The Window Manager draws your application's windows into the Window Manager port. Your application should not draw directly into the Window Manager port, except through custom window definition functions.

Note that the functions `HideFloatingWindows` (page 74) and `ShowFloatingWindows` (page 77) require you to call `InitFloatingWindows` prior to their use.

VERSION NOTES

Supported with Mac OS 8.6 and later.

ShowFloatingWindows

Shows an application's floating windows.

```
pascal OSStatus ShowFloatingWindows (void);
```

function result A result code; see "Result Codes" (page 148). Returns `errFloatingWindowsNotInitialized` (-5609) if you have not called `InitFloatingWindows` prior to `ShowFloatingWindows`; otherwise, returns `noErr` (0).

DISCUSSION

When your application receives a suspend event, it must hide any visible floating windows. When your application receives a resume event, it must make its floating windows visible again. See "Responding to Suspend and Resume Events" (page 50) for an example of how your application can call the `ShowFloatingWindows` function.

SPECIAL CONSIDERATIONS

You must call the function `InitFloatingWindows` (page 75) prior to calling `ShowFloatingWindows`.

The `ShowFloatingWindows` function operates only upon windows created with the `kFloatingWindowClass` constant; see "Window Class Constants" (page 140) for more details on this constant.

VERSION NOTES

Supported with Mac OS 8.6 and later.

SEE ALSO

The function `HideFloatingWindows` (page 74).

TransitionWindow

Displays an animation and plays the theme-appropriate sound for a window when it is shown or hidden.

```
pascal OSStatus TransitionWindow (
                    WindowPtr window,
                    WindowTransitionEffect effect,
                    WindowTransitionAction action,
                    const Rect *rect);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window that is being shown or hidden.

`effect` A value of type `WindowTransitionEffect`. Pass a constant specifying the window transition effect to be performed. With the Mac OS 8.5 Window Manager, the only valid constant is `kWindowZoomTransitionEffect`; see "Window Transition Effect Constant" (page 147) for a description of this value.

`action` A value of type `WindowTransitionAction`. Pass a constant specifying the window transition action to be performed; valid constants are `kWindowShowTransitionAction` and `kWindowHideTransitionAction`. See "Window Transition Action Constants" (page 147) for descriptions of these values.

`rect` A pointer to a structure of type `Rect`.
If you pass `kWindowShowTransitionAction` in the `action` parameter then, before calling `TransitionWindow`, set the rectangle to specify the dimensions and position, in global coordinates, of the area from which the zoom is to start. If you pass `NULL`, `TransitionWindow` uses the center of the display screen as the source rectangle.
If you pass `kWindowHideTransitionAction` in the `action` parameter then, before calling `TransitionWindow`, set the rectangle to specify the dimensions and position, in global coordinates, of the area at which the zoom is to end.
If you pass `NULL`, `TransitionWindow` uses the center of the display screen as the destination rectangle.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `TransitionWindow` function displays an animation of a window's transition between the open and closed states, such as that displayed by the Finder. `TransitionWindow` uses the rectangle specified in the `rect` parameter for one end of the animation (the source or the destination of the zoom, depending upon whether the window is being shown or hidden, respectively) and the window's current size and position for the other end of the animation. `TransitionWindow` also plays sounds appropriate to the current theme for the opening and closing actions. See "Creating a Window" (page 32) for an example of how your application can call the `TransitionWindow` function.

Your application may use `TransitionWindow` instead of the functions `ShowWindow` and `HideWindow`. Like these pre–Mac OS 8.5 Window Manager functions, `TransitionWindow` generates the appropriate update and active events when it shows and hides windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

Accessing Window Information

The Mac OS 8.5 Window Manager provides the following functions for determining information about windows:

- `GetWindowClass` (page 81) obtains the class of a window.
- `GetWindowAttributes` (page 80) obtains the attributes of a window.
- `FrontNonFloatingWindow` (page 80) returns a pointer to the application's frontmost visible window that is not a floating window.
- `IsValidWindowPtr` (page 82) reports whether a pointer is a valid window pointer.

FrontNonFloatingWindow

Returns a pointer to the application's frontmost visible window that is not a floating window.

```
pascal WindowPtr FrontNonFloatingWindow (void);
```

function result A pointer to the first visible window in the window list that is of a nonfloating class. See "Window Class Constants" (page 140) for a description of window classes.

DISCUSSION

Your application should call the `FrontNonFloatingWindow` function when you want to identify the frontmost visible window that is not a floating window. If you want to identify the frontmost visible window, whether floating or not, your application should call the function `FrontWindow`.

GetWindowAttributes

Obtains the attributes of a window.

```
pascal OSStatus GetWindowAttributes (
                    WindowPtr window,
                    WindowAttributes *outAttributes);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose attributes you wish to obtain.

`outAttributes` A pointer to an unsigned 32-bit value of type `WindowAttributes`. On return, the bits are set to the attributes of the specified window. See "Window Attribute Constants" (page 138) for a description of possible attributes. With Mac OS 8.5, if the window was not originally created using the function `CreateNewWindow` (page 67), then all attribute bits are set to 0, and `GetWindowAttributes` returns a `paramErr` (-50) result.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Window attributes specify a window's features (such as whether the window has a close box) and logical attributes (such as whether the window receives update and activate events).

VERSION NOTES

Available with Mac OS 8.5 and later.

GetWindowClass

Obtains the class of a window.

```
pascal OSStatus GetWindowClass (
                    WindowPtr window,
                    WindowClass *outClass);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose class you wish to obtain.

`outClass` A pointer to a value of type `WindowClass`. On return, this value identifies the class of the specified window. See "Window Class Constants" (page 140) for a list of possible window classes. If the window was not originally created using the function `CreateNewWindow` (page 67), the class pointed to by the `outClass` parameter is always identified by the constant `kDocumentWindowClass`.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

A window's class categorizes the window for purposes of display (that is, both the window's appearance and its display ordering) and tracking. A window's class cannot be altered once the window has been created.

VERSION NOTES

Available with Mac OS 8.5 and later.

IsValidWindowPtr

Reports whether a pointer is a valid window pointer.

```
pascal Boolean IsValidWindowPtr (GrafPtr grafPort);
```

`grafPort` A pointer to a graphics port. You pass the pointer to be examined.

function result A value of type `Boolean`. The function returns `true` if the specified pointer is a valid window pointer; otherwise, `false`.

DISCUSSION

A custom control definition may use the `IsValidWindowPtr` function to determine whether it is being asked to draw onscreen or offscreen.

This function is primarily intended for use with debugging your application.

SPECIAL CONSIDERATIONS

The `IsValidWindowPtr` function is a processor-intensive call.

VERSION NOTES

Available with Mac OS 8.5 and later.

Manipulating Window Color Information

The Mac OS 8.5 Window Manager provides the following functions for redrawing a window's content region:

- `SetWindowContentColor` (page 84) sets the color to which a window's content region is redrawn.
- `GetWindowContentColor` (page 83) obtains the color to which a window's content region is redrawn.
- `SetWindowContentPattern` (page 85) sets the pattern to which a window's content region is redrawn.
- `GetWindowContentPattern` (page 83) obtains the pattern to which a window's content region is redrawn.

GetWindowContentColor

Obtains the color to which a window's content region is redrawn.

```
pascal OSStatus GetWindowContentColor (
                    WindowPtr window,
                    RGBColor *color);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose content color is being retrieved.

`color` A pointer to an `RGBColor` structure. On return, the structure contains the content color for the specified window.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `GetWindowContentColor` function obtains the color to which the window's content region is redrawn. See "Window Content Color" (page 24) for further discussion.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `SetWindowContentColor` (page 84).

GetWindowContentPattern

Obtains the pattern to which a window's content region is redrawn.

```
pascal OSStatus GetWindowContentPattern (
                    WindowPtr window,
                    PixPatHandle outPixPat);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose content pattern is being retrieved.

`outPixPat`	A handle to a structure of type `PixPat`. On return, the structure contains a copy of the content pattern data for the specified window, which your application is responsible for disposing.
function result	A result code. See “Result Codes” (page 148).

DISCUSSION

The `GetWindowContentPattern` function obtains the pattern to which the window’s content region is redrawn. See “Window Content Color” (page 24) for further discussion.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `SetWindowContentPattern` (page 85).

SetWindowContentColor

Sets the color to which a window’s content region is redrawn.

```
pascal OSStatus SetWindowContentColor (
                    WindowPtr window,
                    RGBColor *color);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window whose content color is being set.
`color`	A pointer to an `RGBColor` structure. Before calling `SetWindowContentColor`, set this structure to specify the content color to be used.
function result	A result code. See “Result Codes” (page 148).

DISCUSSION

If your application uses the `SetWindowContentColor` function, the window’s content region is redrawn to the color you specify, without affecting the value

specified in the window's `CGrafPort` structure for the current background color. Applications should use `SetWindowContentColor` instead of the `SetWinColor` function. See "Drawing in a Window's Content Region" (page 42) for further discussion and an example of how your application might call the `SetWindowContentColor` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `GetWindowContentColor` (page 83).

SetWindowContentPattern

Sets the pattern to which a window's content region is redrawn.

```
pascal OSStatus SetWindowContentPattern (
                    WindowPtr window,
                    PixPatHandle pixPat);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose content pattern is being set.

`pixPat` A handle to a structure of type `PixPat`. Before calling `SetWindowContentPattern`, set this structure to specify the content pattern to be used. This handle is copied by the Window Manager, and your application continues to own the original. Therefore there may be higher RAM requirements for applications with numerous identically patterned windows.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

If your application uses the `SetWindowContentPattern` function, the window's content region is redrawn to the pattern you specify, without affecting the value specified in the window's `CGrafPort` structure for the current background pattern. See "Drawing in a Window's Content Region" (page 42) for further

discussion and an example of calling the related function `SetWindowContentColor` (page 84).

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `GetWindowContentPattern` (page 83).

Zooming Windows

The Mac OS 8.5 Window Manager provides the following functions for zooming windows:

- `ZoomWindowIdeal` (page 90) zooms a window in accordance with human interface guidelines.
- `IsWindowInStandardState` (page 87) determines whether a window is currently zoomed in to the user state or zoomed out to the standard state.
- `SetWindowIdealUserState` (page 89) sets the size and position of a window in its user state.
- `GetWindowIdealUserState` (page 86) obtains the size and position of a window in its user state.

GetWindowIdealUserState

Obtains the size and position of a window in its user state.

```
pascal OSStatus GetWindowIdealUserState (
                    WindowPtr window,
                    Rect *userState);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which you wish to obtain the user state.

`userState`	A pointer to a structure of type `Rect`. On return, this rectangle specifies the current size and position of the window's user state, in global coordinates.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

Because the window definition function relies upon the `WStateData` structure, it is unaware of the ideal standard state, and this causes the user state data that it stores in the `WStateData` structure to be unreliable. While the Window Manager is reliably aware of the window's zoom state, it cannot record the current user state in the `WStateData` structure, because the window definition function can overwrite that data. Therefore, the function `ZoomWindowIdeal` (page 90) maintains the window's user state independently of the `WStateData` structure. The `GetWindowIdealUserState` function gives your application access to the user state data maintained by `ZoomWindowIdeal`. However, your application should not typically need to use this function; it is supplied for completeness.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `SetWindowIdealUserState` (page 89).

IsWindowInStandardState

Determines whether a window is currently zoomed in to the user state or zoomed out to the standard state.

```
pascal Boolean IsWindowInStandardState (
                    WindowPtr window,
                    Point *idealSize,
                    Rect *idealStandardState);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window for which you wish to determine the zoom state.

`idealSize` A pointer to a structure of type `Point`. Before calling `IsWindowInStandardState`, set the `Point` structure to contain the ideal width and height of the window's content region, regardless of the actual screen device dimensions. If you set `idealSize` to `NULL`, `IsWindowInStandardState` examines the dimensions stored in the `stdState` field of the `WStateData` structure.

`idealStandardState`
A pointer to a structure of type `Rect`. On return, the rectangle contains the global coordinates for the content region of the window in its standard state, based on the data supplied in the `idealSize` parameter. You may pass `NULL` if you do not wish to receive this data.

function result A value of type `Boolean`. The `IsWindowInStandardState` function returns `true` if the window is currently in its standard state; otherwise, if the window is currently in the user state, `IsWindowInStandardState` returns `false`.

DISCUSSION

The `IsWindowInStandardState` function compares the window's current dimensions to those referred to by the `idealSize` parameter to determine if the window is currently in the standard state. Your application may use `IsWindowInStandardState` to decide whether a user's click of the zoom box is a request to zoom to the user state or the standard state, as described in the function `ZoomWindowIdeal` (page 90). Your application may also use `IsWindowInStandardState` to determine the size and position of the standard state that the Window Manager would calculate for a window, given a specified ideal size; this value is produced in the `idealStandardState` parameter. See "Zooming a Window Gracefully" (page 54) for an example of calling the `IsWindowInStandardState` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

SetWindowIdealUserState

Sets the size and position of a window in its user state.

```
pascal OSStatus SetWindowIdealUserState (
                    WindowPtr window,
                    Rect *userState);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which you wish to set the user state.

`userState` A pointer to a structure of type `Rect`. Before calling `SetWindowIdealUserState` set this rectangle to specify the new size and position of the window's user state, in global coordinates.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Because the window definition function relies upon the `WStateData` structure, it is unaware of the ideal standard state, and this causes the user state data that it stores in the `WStateData` structure to be unreliable. While the Window Manager is reliably aware of the window's zoom state, it cannot record the current user state in the `WStateData` structure, because the window definition function can overwrite that data. Therefore, the function `ZoomWindowIdeal` (page 90) maintains the window's user state independently of the `WStateData` structure. The `SetWindowIdealUserState` function gives your application access to the user state data maintained by `ZoomWindowIdeal`. However, your application does not typically need to use this function; it is supplied for completeness.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `GetWindowIdealUserState` (page 86).

ZoomWindowIdeal

Zooms a window in accordance with human interface guidelines.

```
pascal OSStatus ZoomWindowIdeal (
                    WindowPtr window,
                    SInt16 partCode,
                    Point *ioIdealSize);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window to be zoomed.

`partCode` A value specifying the direction of the zoom being requested. Your application passes in the relevant value (either the `inZoomIn` or the `inZoomOut` constant).

`ioIdealSize` A pointer to a structure of type `Point`.
When you specify `inZoomIn` in the `partCode` parameter, you pass a pointer to the `Point` structure, but do not fill the structure with data. On return, the `Point` structure contains the new height and width of the window's content region, and `ZoomWindowIdeal` restores the previous user state.
When you specify `inZoomOut` in the `partCode` parameter, you pass the ideal height and width of the window's content region in the `Point` structure. On return, the `Point` structure contains the new height and width of the window's content region; `ZoomWindowIdeal` saves the user state of the window and zooms the window to its ideal size for the standard state.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Applications should use the `ZoomWindowIdeal` function instead of the older function `ZoomWindow`. When your application calls `ZoomWindowIdeal`, it automatically conforms to the human interface guidelines for determining a window's standard state, as described in "Window Zooming" (page 21).

The `ZoomWindowIdeal` function calculates a window's ideal standard state and updates a window's ideal user state independently of the `WStateData` structure. Previously, the window definition function was responsible for updating the user state, but because it relies upon the `WStateData` structure, the window definition function is unaware of the ideal standard state and can no longer track the window's zoom state reliably.

While the Window Manager is reliably aware of the window's zoom state, it cannot record the current user state in the `WStateData` structure, because the window definition function can overwrite that data. Therefore, if your application uses `ZoomWindowIdeal`, the `WStateData` structure is superseded, and the result of the `FindWindow` function should be ignored when determining whether a particular user click of the zoom box is a request to zoom in or out. When you adopt `ZoomWindowIdeal` and your application receives a result of either `inZoomIn` or `inZoomOut` from `FindWindow`, your application must use the function `IsWindowInStandardState` (page 87) and code such as that in Listing 2-9 in "Zooming a Window Gracefully" (page 54) to determine the appropriate part code to pass in the `partCode` parameter.

VERSION NOTES

Available with Mac OS 8.5 and later.

Sizing and Positioning Windows

The Mac OS 8.5 Window Manager provides the following functions for working with the size and position of windows:

- `SetWindowBounds` (page 96) sets a window's size and position from the bounding rectangle of the specified window region.
- `GetWindowBounds` (page 92) obtains the size and position of the bounding rectangle of the specified window region.
- `MoveWindowStructure` (page 93) positions a window relative to its structure region.
- `ResizeWindow` (page 95) handles all user interaction while a window is being resized.
- `RepositionWindow` (page 94) positions a window relative to another window or a display screen.

GetWindowBounds

Obtains the size and position of the bounding rectangle of the specified window region.

```
pascal OSStatus GetWindowBounds (
                    WindowPtr window,
                    WindowRegionCode regionCode,
                    Rect *globalBounds);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose bounds you wish to obtain.

`regionCode` A value of type `WindowRegionCode`. Pass in a constant identifying the window region whose bounds you wish to obtain. Currently, the only valid values for the region code are `kWindowStructureRgn` and `kWindowContentRgn`; see *Mac OS 8 Window Manager Reference* for descriptions of these and other `WindowRegionCode` constants.

`globalBounds` A pointer to a structure of type `Rect`. On return, the rectangle contains the dimensions and position, in global coordinates, of the window region specified in the `regionCode` parameter.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

When you call the function `SetWindowBounds` (page 96), your application specifies whether the window's content region or its structure region is more important in determining the window's ultimate size and position. This distinction can be important with versions of the Mac OS running the Appearance Manager, since the total dimensions of a window—and, therefore, its spatial relationship to the rest of the screen—may vary from appearance to appearance. Use the `GetWindowBounds` function to obtain the bounding rectangle for either of these regions for the specified window.

VERSION NOTES

Available with Mac OS 8.5 and later.

MoveWindowStructure

Positions a window relative to its structure region.

```
pascal OSStatus MoveWindowStructure (
                    WindowPtr window,
                    short hGlobal,
                    short vGlobal);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window to be moved.

`hGlobal` Pass a value specifying the horizontal position, in global coordinates, to which the left edge of the window's structure region is to be moved.

`vGlobal` Pass a value specifying the vertical position, in global coordinates, to which the top edge of the window's structure region is to be moved.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `MoveWindowStructure` function moves the specified window, but does not change the window's size. When your application calls `MoveWindowStructure`, the positioning of the specified window is determined by the positioning of its structure region. This is in contrast to the `MoveWindow` function, where the positioning of the window's content region determines the positioning of the window. After moving the window, `MoveWindowStructure` displays the window in its new position.

Note that your application should not call the `MoveWindowStructure` function to position a window when the user drags the window by its drag region. When the user drags the window, your application should call the pre–Mac OS 8.5 Window Manager function `DragWindow`.

VERSION NOTES

Available with Mac OS 8.5 and later.

RepositionWindow

Positions a window relative to another window or a display screen.

```
pascal OSStatus RepositionWindow (
                    WindowPtr window,
                    WindowPtr parentWindow,
                    WindowPositionMethod method);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose position you want to set.

`parentWindow` A value of type `WindowPtr`. Pass a pointer to the "parent" window, as defined by your application. In cases where the window positioning method does not require a parent window, you should set the `parentWindow` parameter to `NULL`.

`method` A value of type `WindowPositionMethod`. Pass a constant specifying the window positioning method to be used; see "RepositionWindow Constants" (page 136) for descriptions of possible values.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Your application may call the `RepositionWindow` function to position any window, relative to another window or to a display screen. After positioning the window, `RepositionWindow` displays the window in its new position. See "Positioning a Window on the Desktop" (page 35) for further discussion.

VERSION NOTES

Available with Mac OS 8.5 and later.

ResizeWindow

Handles all user interaction while a window is being resized.

```
pascal Boolean ResizeWindow (
                    WindowPtr window,
                    Point startPoint,
                    const Rect *sizeConstraints,
                    Rect *newContentRect);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window to be resized.

`startPt` A structure of type `Point`. Before calling `ResizeWindow`, your application should set the `Point` structure to contain the location, specified in global coordinates, where the mouse-down event occurred. Your application may retrieve this value from the `where` field of the `event` structure.

`sizeConstraints`
A pointer to a structure of type `Rect`. Before calling `ResizeWindow`, set the rectangle to specify the limits on the vertical and horizontal measurements of the content rectangle, in pixels. Although this parameter gives the address of a structure that is in the form of the `Rect` data type, the four numbers in the structure represent limits, not screen coordinates. The `top`, `left`, `bottom`, and `right` fields of the structure specify the minimum vertical measurement (`top`), the minimum horizontal measurement (`left`), the maximum vertical measurement (`bottom`), and the maximum horizontal measurement (`right`). The minimum dimensions should be large enough to allow a manageable rectangle; 64 pixels on a side is typical. The maximum dimensions can be no greater than 32,767. You can pass `NULL` to allow the user to resize the window to any size that is contained onscreen.

`newContentRect`
A pointer to a structure of type `Rect`. On return, the structure contains the new dimensions of the window's content region, in global coordinates.

function result A value of type `Boolean`. The function returns `true` if the window was successfully resized; otherwise, `false`.

DISCUSSION

The `ResizeWindow` function moves an outline (grow image) of the window's edges around the screen, following the user's cursor movements, and handles all user interaction until the mouse button is released. Unlike with the function `GrowWindow`, there is no need to follow this call with a call to the function `SizeWindow`, because once the mouse button is released, `ResizeWindow` resizes the window if the user has changed the window size. Once the resizing is complete, `ResizeWindow` draws the window in the new size.

Your application should call the `ResizeWindow` function instead of the earlier Window Manager functions `SizeWindow` and `GrowWindow`. Some appearances may allow the window to be resized from any corner, not just the bottom right, and as a result, when the user resizes the window, the window may move on the screen and not merely change size. `ResizeWindow` informs your application of the new window bounds, so that your application can respond to any changes in the window's position.

VERSION NOTES

Supported with Mac OS 8.6 and later.

SetWindowBounds

Sets a window's size and position from the bounding rectangle of the specified window region.

```
pascal OSStatus SetWindowBounds (
                    WindowPtr window,
                    WindowRegionCode regionCode,
                    const Rect *globalBounds);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose bounds are to be set.

`regionCode` A value of type `WindowRegionCode`. Pass in a constant specifying the region to be used in determining the window's size and position. With Mac OS 8.5, the only valid values for the region code are `kWindowStructureRgn` and `kWindowContentRgn`; see *Mac OS 8 Window Manager Reference* for descriptions of these and other `WindowRegionCode` constants.

`globalBounds`	A pointer to a structure of type `Rect`. Before calling `SetWindowBounds`, set the rectangle to specify the dimensions and position, in global coordinates, of the window region specified in the `regionCode` parameter.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

The `SetWindowBounds` function sets a window's size and position to that specified by the rectangle that your application passes in the `globalBounds` parameter. After doing so, `SetWindowBounds` displays the window.

When you call the `SetWindowBounds` function, your application specifies whether the window's content region or its structure region is more important in determining the window's ultimate size and position. This distinction can be important with versions of the Mac OS running the Appearance Manager, since the total dimensions of a window—and, therefore, its spatial relationship to the rest of the screen—may vary from appearance to appearance. In general, you should specify `kWindowStructureRgn` for the `regionCode` parameter if how the window as a whole relates to a given monitor is more important than the exact positioning of its content on the screen. On the other hand, if you specify `kWindowContentRgn` for the `regionCode` parameter because the positioning of your application's content is of greatest concern, then it is important to note that with some appearances some part of the window's structure region or "frame" may extend past the edge of a monitor and not be displayed.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `GetWindowBounds` (page 92).

Establishing Proxy Icons

The Mac OS 8.5 Window Manager provides the following functions for establishing proxy icons:

- `SetWindowProxyFSSpec` (page 104) associates a file with a window.

- `GetWindowProxyFSSpec` (page 99) obtains a file system specification structure for the file that is associated with a window.
- `SetWindowProxyAlias` (page 101) associates a file with a window.
- `GetWindowProxyAlias` (page 98) obtains an alias for the file that is associated with a window.
- `SetWindowProxyCreatorAndType` (page 102) sets the proxy icon for a window that lacks an associated file.
- `SetWindowProxyIcon` (page 105) overrides the default proxy icon for a window.
- `GetWindowProxyIcon` (page 100) obtains a window's proxy icon.
- `RemoveWindowProxy` (page 100) dissociates a file from a window.

GetWindowProxyAlias

Obtains an alias for the file that is associated with a window.

```
pascal OSStatus GetWindowProxyAlias (
                    WindowPtr window,
                    AliasHandle *alias);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which you wish to determine the associated file.

`alias` A pointer to a value of type `AliasHandle`. On return, the `AliasRecord` structure referenced by the alias handle contains a copy of the alias data for the file associated with the specified window. Your application must dispose of this handle. See *Inside Macintosh: Files* for more information on aliases.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Your application can call the `GetWindowProxyAlias` function to retrieve alias data for the file associated with a window. See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `SetWindowProxyAlias` (page 101).

GetWindowProxyFSSpec

Obtains a file system specification structure for the file that is associated with a window.

```
pascal OSStatus GetWindowProxyFSSpec (
                    WindowPtr window,
                    FSSpec *outFile);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which you wish to determine the associated file.

`outFile` A pointer to an `FSSpec` structure. On return, this structure contains a copy of the file system specification data for the file associated with the specified window. See *Inside Macintosh: Files* for more information on the `FSSpec` data type.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

You can use the `GetWindowProxyFSSpec` function to obtain identifying information about a proxy file: its volume reference number, directory ID, and file name. See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `SetWindowProxyFSSpec` (page 104).

GetWindowProxyIcon

Obtains a window's proxy icon.

```
pascal OSStatus GetWindowProxyIcon (
                    WindowPtr window,
                    IconRef *outIcon);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window for which you wish to obtain the proxy icon.
`outIcon`	A pointer to a value of type `IconRef`. On return, the icon reference identifies the icon currently used for the window's proxy icon. Your application is responsible for disposing of this icon reference.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

Your application should call the `GetWindowProxyIcon` function if it needs to obtain an `IconRef` value for a proxy icon, such as is required for the function `SetWindowProxyIcon` (page 105). See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

RemoveWindowProxy

Dissociates a file from a window.

```
pascal OSStatus RemoveWindowProxy (WindowPtr window);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window for which you wish to remove the associated file.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

The `RemoveWindowProxy` function redraws the window title bar after removing all data associated with a given file, including the proxy icon, path menu, and file data.

SPECIAL CONSIDERATIONS

With Mac OS 8.5, you must save and restore the current graphics port—by calling the QuickDraw functions `GetPort` and `SetPort`—around each call to the `RemoveWindowProxy` function. See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

SetWindowProxyAlias

Associates a file with a window.

```
pascal OSStatus SetWindowProxyAlias (
                    WindowPtr window,
                    AliasHandle alias);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window with which the specified file is to be associated.

`alias` A value of type `AliasHandle`. Pass in a handle to a structure of type `AliasRecord` for the file to associate with the specified window. You can obtain an alias handle by calling the function `GetWindowProxyAlias` (page 98). See *Inside Macintosh: Files* for more information on aliases.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Your application should call the `SetWindowProxyAlias` function to establish a proxy icon for a given window. The creator code and file type of the file

associated with a window determine the proxy icon that is displayed for the window.

Because the `SetWindowProxyAlias` function won't work without a saved file, you must establish the initial proxy icon for a new, untitled window with the function `SetWindowProxyCreatorAndType` (page 102), which requires that you know the file type and creator code for the file, but does not require that the file have been saved.

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

SPECIAL CONSIDERATIONS

With Mac OS 8.5, you must save and restore the current graphics port—by calling the QuickDraw functions `GetPort` and `SetPort`—around each call to the `SetWindowProxyAlias` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `SetWindowProxyFSSpec` (page 104).

SetWindowProxyCreatorAndType

Sets the proxy icon for a window that lacks an associated file.

```
pascal OSStatus SetWindowProxyCreatorAndType (
                    WindowPtr window,
                    OSType fileCreator,
                    OSType fileType,
                    SInt16 vRefNum);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which you want to set the proxy icon.

`fileCreator`	A four-character code. Pass in the code that is to be used, together with the `fileType` parameter, to determine the proxy icon. This typically is the creator code of the file that would be created, were the user to save the contents of the window.
`fileType`	A four-character code. Pass in a code that is to be used, together with the `fileCreator` parameter, to determine the proxy icon. This typically is the file type of the file that would be created, were the user to save the contents of the window.
`vRefNum`	A value identifying the volume containing the default desktop database to search for the icon associated with the file type and creator code specified in the `fileCreator` and `fileType` parameters. Pass `kOnSystemDisk` if the volume is unknown.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

A new, untitled window needs a proxy icon in order to maintain visual consistency with other windows under Mac OS 8.5 and later. Your application should call the `SetWindowProxyCreatorAndType` function when you want to establish a proxy icon for a window, but the window's data has not yet been saved to a file. See "Creating a Window" (page 32) for an example of how your application can call the `SetWindowProxyCreatorAndType` function.

If the window's data has been saved to a file, your application can call the functions `SetWindowProxyFSSpec` (page 104) or `SetWindowProxyAlias` (page 101) to associate the file with the window and thereby establish the proxy icon. See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

SPECIAL CONSIDERATIONS

With Mac OS 8.5, you must save and restore the current graphics port—by calling the QuickDraw functions `GetPort` and `SetPort`—around each call to the `SetWindowProxyCreatorAndType` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

SetWindowProxyFSSpec

Associates a file with a window.

```
pascal OSStatus SetWindowProxyFSSpec (
                    WindowPtr window,
                    const FSSpec *inFile);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window with which the specified file is to be associated.

`inFile` A pointer to an `FSSpec` structure. Before calling `SetWindowProxyFSSpec`, set the file system specification structure to contain the data for the file to associate with the specified window. You can obtain an `FSSpec` structure by calling the function `GetWindowProxyFSSpec` (page 99). See *Inside Macintosh: Files* for more information on the `FSSpec` data type.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Your application should call the `SetWindowProxyFSSpec` function to establish a proxy icon for a given window. The creator code and file type of the file associated with a window determine the proxy icon that is displayed for the window.

Because the `SetWindowProxyFSSpec` function won't work without a saved file, you must establish the initial proxy icon for a new, untitled window with the function `SetWindowProxyCreatorAndType` (page 102), which requires that you know the file type and creator code for the file, but does not require that the file have been saved.

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

SPECIAL CONSIDERATIONS

With Mac OS 8.5, you must save and restore the current graphics port—by calling the QuickDraw functions `GetPort` and `SetPort`—around each call to the `SetWindowProxyFSSpec` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `SetWindowProxyAlias` (page 101).

SetWindowProxyIcon

Overrides the default proxy icon for a window.

```
pascal OSStatus SetWindowProxyIcon (
                    WindowPtr window,
                    IconRef icon);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which you wish to set the proxy icon.

`icon` A value of type `IconRef`. Pass an icon reference identifying the icon to be used for the window's proxy icon. If there is already a proxy icon in use of the type desired, an `IconRef` value may be obtained for that icon by calling the function `GetWindowProxyIcon` (page 100). Otherwise, your application must call the Icon Services function `GetIconRefFromFile` to get a value of type `IconRef`.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

If you wish to override the proxy icon that the Window Manager displays by default for a given file, your application should call the `SetWindowProxyIcon` function.

More typically, when you do not wish to override a window's default proxy icon, your application would call one of the following functions: `SetWindowProxyFSSpec` (page 104), `SetWindowProxyAlias` (page 101), or `SetWindowProxyCreatorAndType` (page 102).

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

SPECIAL CONSIDERATIONS

With Mac OS 8.5, you must save and restore the current graphics port—by calling the QuickDraw functions `GetPort` and `SetPort`—around each call to the `SetWindowProxyIcon` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `GetWindowProxyIcon` (page 100).

Coordinating Proxy Icons With Drag-and-Drop Management

The Mac OS 8.5 Window Manager provides the following functions for dragging proxy icons:

- `TrackWindowProxyDrag` (page 112) handles all aspects of the drag process when the user drags a proxy icon.
- `TrackWindowProxyFromExistingDrag` (page 113) allows custom handling of the drag process when the user drags a proxy icon.
- `BeginWindowProxyDrag` (page 107) creates the drag reference and the drag image when the user drags a proxy icon.
- `EndWindowProxyDrag` (page 108) disposes of the drag reference when the user completes the drag of a proxy icon.

The Mac OS 8.5 Window Manager provides the following functions for indicating whether a proxy icon can currently be dragged:

- `SetWindowModified` (page 111) sets the modification state of the specified window.
- `IsWindowModified` (page 111) obtains the modification state of the specified window.

The Mac OS 8.5 Window Manager provides the following function for indicating whether a window is a valid drag-and-drop target:

- HiliteWindowFrameForDrag (page 110) sets the highlight state of the window's structure region to reflect the window's validity as a drag-and-drop destination.

BeginWindowProxyDrag

Creates the drag reference and the drag image when the user drags a proxy icon.

```
pascal OSStatus BeginWindowProxyDrag (
                    WindowPtr window,
                    DragReference *outNewDrag,
                    RgnHandle outDragOutlineRgn);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose proxy icon is being dragged.

`outNewDrag` A pointer to a value of type `DragReference`. On return, the value refers to the current drag process.

`outDragOutlineRgn`
A value of type `RgnHandle`. Your application can create this handle with a call to the QuickDraw function `NewRgn`. On return, this region is set to the outline of the icon being dragged.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Typically, if the proxy icon represents a type of object (currently, file system entities such as files, folders, and volumes) supported by the Window Manager, the Window Manager can handle all aspects of the drag process itself, and your application should call the function `TrackWindowProxyDrag` (page 112). However, if the proxy icon represents a type of data that the Window Manager does not support, or if you wish to implement custom dragging behavior, your application should call the function `TrackWindowProxyFromExistingDrag` (page 113).

The `TrackWindowProxyFromExistingDrag` function accepts an existing drag reference and adds file data if the window contains a file proxy. If your application uses `TrackWindowProxyFromExistingDrag`, you then have the choice of using this function in conjunction with the functions `BeginWindowProxyDrag` and

`EndWindowProxyDrag` (page 108) or simply calling `TrackWindowProxyFromExistingDrag` and handling all aspects of creating and disposing of the drag yourself.

Specifically, your application can call `BeginWindowProxyDrag` to set up the drag image and drag reference. Your application must then track the drag, using `TrackWindowProxyFromExistingDrag`, and do any required moving of data and, finally, call `EndWindowProxyDrag` to dispose of the drag reference. `BeginWindowProxyDrag` should not be used for types handled by the Window Manager unless the application wishes to implement custom dragging behavior for those types.

Your application detects a drag when the function `FindWindow` returns the `inProxyIcon` result code; see "FindWindow Result Code Constant for the Proxy Icon" (page 135) for more details.

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

EndWindowProxyDrag

Disposes of the drag reference when the user completes the drag of a proxy icon.

```
pascal OSStatus EndWindowProxyDrag (
                    WindowPtr window,
                    DragReference theDrag);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose proxy icon is being dragged.

`theDrag` A value of type `DragReference` that refers to the current drag process. Pass in the value produced in the `outNewDrag` parameter of `BeginWindowProxyDrag`.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Typically, if the proxy icon represents a type of object (currently, file system entities such as files, folders, and volumes) supported by the Window Manager, the Window Manager can handle all aspects of the drag process itself, and your application should call the function `TrackWindowProxyDrag` (page 112). However, if the proxy icon represents a type of data that the Window Manager does not support, or if you wish to implement custom dragging behavior, your application should call the function `TrackWindowProxyFromExistingDrag` (page 113).

The `TrackWindowProxyFromExistingDrag` function accepts an existing drag reference and adds file data if the window contains a file proxy. If your application uses `TrackWindowProxyFromExistingDrag`, you then have the choice of using this function in conjunction with the functions `BeginWindowProxyDrag` (page 107) and `EndWindowProxyDrag` or simply calling `TrackWindowProxyFromExistingDrag` and handling all aspects of creating and disposing of the drag yourself.

Specifically, your application can call `BeginWindowProxyDrag` to set up the drag image and drag reference. Your application must then track the drag, using `TrackWindowProxyFromExistingDrag`, and do any required moving of data and, finally, call `EndWindowProxyDrag` to dispose of the drag reference and its associated image data. The `EndWindowProxyDrag` function does not dispose of the region created for use by `BeginWindowProxyDrag`, however, so this remains the application's responsibility to dispose. The `EndWindowProxyDrag` function should not be used for types handled by the Window Manager unless the application wishes to implement custom dragging behavior for those types.

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

HiliteWindowFrameForDrag

Sets the highlight state of the window's structure region to reflect the window's validity as a drag-and-drop destination.

```
pascal OSStatus HiliteWindowFrameForDrag (
                    WindowPtr window,
                    Boolean hilited);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which you wish to set the highlight state.

`hilited` A value of type `Boolean`. Set to `true` to indicate that the window's frame should be highlighted; otherwise, `false`.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Applications typically call the Drag Manager functions `ShowDragHilite` and `HideDragHilite` to indicate that a window is a valid drag-and-drop destination. If your application does not do this—that is, if your application implements any type of custom drag highlighting, such as highlighting more than one area of a window at a time—it must call the `HiliteWindowFrameForDrag` function.

The `HiliteWindowFrameForDrag` function highlights a window's proxy icon when the user drags content inside the window that is a valid content type for that destination. The default behavior of system-defined windows is to highlight the proxy icon along with the window's content area when the window is a valid drag-and-drop destination. If you call the Drag Manager functions `ShowDragHilite` and `HideDragHilite`, you don't need to use `HiliteWindowFrameForDrag`.

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

IsWindowModified

Obtains the modification state of the specified window.

```
pascal Boolean IsWindowModified (WindowPtr window);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose modification state is to be obtained.

function result A value of type `Boolean`. The function returns `true` to indicate that the content of the window has been modified; otherwise, `false`. Newly created windows start out with their modification state automatically set to `true`.

DISCUSSION

Your application can use the functions `IsWindowModified` and `SetWindowModified` (page 111) instead of maintaining its own separate record of the modification state of the content of a window.

VERSION NOTES

Available with Mac OS 8.5 and later.

SetWindowModified

Sets the modification state of the specified window.

```
pascal OSStatus SetWindowModified (
                    WindowPtr window,
                    Boolean modified);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose modification state is to be set.

`modified` A value of type `Boolean`. Set to `true` to indicate that the content of the window has been modified; otherwise, set to `false`.

function result A result code. See “Result Codes” (page 148).

DISCUSSION

Your application can use the functions `SetWindowModified` and `IsWindowModified` (page 111) instead of maintaining its own separate record of the modification state of the content of a window.

Your application should distinguish between the modification state of the window and the modification state of the window's contents, typically a document. The modification state of the window contents are what should affect `SetWindowModified`. For example, in the case of a word processing document, you call `SetWindowModified` (passing `true` in the `modified` parameter) whenever the user types new characters into the document. However, you do not call `SetWindowModified` when the user moves the window, because that change does not affect the document contents. If you need to track whether the window position has changed, you need to do this with your own flag.

See "Setting a Window's Modification State" (page 57) for an example of how your application might call the `SetWindowModified` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

TrackWindowProxyDrag

Handles all aspects of the drag process when the user drags a proxy icon.

```
pascal OSStatus TrackWindowProxyDrag (
                    WindowPtr window,
                    Point startPt);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window whose proxy icon is being dragged.

`startPt` A structure of type `Point`. Before calling `TrackWindowProxyDrag`, your application should set the `Point` structure to contain the point, specified in global coordinates, where the mouse-down event that began the drag occurred. Your application may retrieve this value from the `where` field of the `event` structure.

function result A result code. See "Result Codes" (page 148). If you receive the error `errUserWantsToDragWindow` (–5607), your application should respond by calling the Window Manager function `DragWindow`. Errors are also returned from the Drag Manager, including `userCanceledErr` (–128).

DISCUSSION

If your application uses proxy icons to represent a type of object (currently, file system entities such as files, folders, and volumes) supported by the Window Manager, your application should call the `TrackWindowProxyDrag` function, and the Window Manager can handle all aspects of the drag process for you. If your application calls the `TrackWindowProxyDrag` function, it does not have to call the Drag Manager function `WaitMouseMoved` before starting to track the drag, as the Window Manager handles this automatically. However, if a proxy icon represents a type of data that the Window Manager does not support, or if you wish to implement custom dragging behavior, your application should call the function `TrackWindowProxyFromExistingDrag` (page 113).

Your application detects that a user is dragging one of its proxy icons when the function `FindWindow` returns the `inProxyIcon` result code; see "FindWindow Result Code Constant for the Proxy Icon" (page 135) for more details.

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

TrackWindowProxyFromExistingDrag

Allows custom handling of the drag process when the user drags a proxy icon.

```
pascal OSStatus TrackWindowProxyFromExistingDrag (
                    WindowPtr window,
                    Point startPt,
                    DragReference drag,
                    RgnHandle inDragOutlineRgn);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window whose proxy icon is being dragged.
`startPt`	A structure of type `Point`. Before calling `TrackWindowProxyFromExistingDrag`, your application should set the `Point` structure to contain the point, specified in global coordinates, where the mouse-down event that began the drag occurred. Your application may retrieve this value from the `where` field of the `event` structure.
`drag`	A value of type `DragReference` that refers to the current drag process. Pass in the value produced in the `outNewDrag` parameter of the function `BeginWindowProxyDrag` (page 107). If you are not using `BeginWindowProxyDrag` in conjunction with `TrackWindowProxyFromExistingDrag`, you must create the drag reference yourself with the Drag Manager function `NewDrag`.
`inDragOutlineRgn`	A value of type `RgnHandle`. Pass in a region handle representing an outline of the icon being dragged. You may obtain a handle to this region from the `outDragOutlineRgn` parameter of `BeginWindowProxyDrag`. If you are not using `BeginWindowProxyDrag` in conjunction with `TrackWindowProxyFromExistingDrag`, you must create the region yourself.
function result	A result code. See "Result Codes" (page 148). Errors are also returned from the Drag Manager, including `userCanceledErr` (-128).

DISCUSSION

Typically, if the proxy icon represents a type of object (currently, file system entities such as files, folders, and volumes) supported by the Window Manager, the Window Manager can handle all aspects of the drag process itself, and your application should call the function `TrackWindowProxyDrag` (page 112). However, if the proxy icon represents a type of data that the Window Manager does not support, or if you wish to implement custom dragging behavior, your application should call the `TrackWindowProxyFromExistingDrag` function.

The `TrackWindowProxyFromExistingDrag` function accepts an existing drag reference and adds file data if the window contains a file proxy. If your application uses `TrackWindowProxyFromExistingDrag`, you then have the choice of

using this function in conjunction with the functions `BeginWindowProxyDrag` (page 107) and `EndWindowProxyDrag` (page 108) or simply calling `TrackWindowProxyFromExistingDrag` and handling all aspects of creating and disposing of the drag yourself.

Your application detects a drag when the function `FindWindow` returns the `inProxyIcon` result code; see "FindWindow Result Code Constant for the Proxy Icon" (page 135) for more details.

See "Supporting Window Proxy Icons" (page 37) for examples of how your application can provide proxy icon support in its document windows.

VERSION NOTES

Available with Mac OS 8.5 and later.

Activating Window Path Pop-Up Menus

The Mac OS 8.5 Window Manager provides the following functions for handling the activation of window path pop-up menus:

- `IsWindowPathSelectClick` (page 115) reports whether a mouse click should activate the window path pop-up menu.
- `WindowPathSelect` (page 116) displays a window path pop-up menu.

IsWindowPathSelectClick

Reports whether a mouse click should activate the window path pop-up menu.

```
pascal Boolean IsWindowPathSelectClick (
                    WindowPtr window,
                    EventRecord *event);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window in which the mouse-down event occurred.

`event` A pointer to a value of type `EventRecord`. Pass a pointer to the `EventRecord` structure containing the mouse-down event that `IsWindowPathSelectClick` is to examine.

function result A value of type `Boolean`. The function returns `true` if the mouse click should activate the window path pop-up menu; otherwise, `false`.

DISCUSSION

The Mac OS 8.5 Window Manager provides system support for your application to display window path pop-up menus—like those used in Finder windows. When the user presses the Command key and clicks on the window's title, the window displays a pop-up menu containing a standard file system path, informing the user of the location of the document displayed in the window and allowing the user to open windows for folders along the path.

Because the window title includes both the proxy icon region and part of the drag region of the window, your application must be prepared to respond to a click in either region by displaying a window path pop-up menu. Therefore, when the `FindWindow` function returns either the `inDrag` or the `inProxyIcon` result code—you should pass the event to the `IsWindowPathSelectClick` function to determine whether the mouse-down event should activate the window path pop-up menu. If `IsWindowPathSelectClick` returns a value of `true`, your application should then call the function `WindowPathSelect` (page 116) to display the menu. Listing 2-5 in "Displaying a Window Path Pop-Up Menu" (page 46) shows how your application might handle a user request to display the window path pop-up menu.

VERSION NOTES

Available with Mac OS 8.5 and later.

WindowPathSelect

Displays a window path pop-up menu.

```
pascal OSStatus WindowPathSelect (
                    WindowPtr window,
                    MenuHandle menu,
                    SInt32 *outMenuResult);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window for which a window path pop-up menu is to be displayed.

menu	A value of type MenuHandle. Pass a handle to a menu to be displayed for the specified window or NULL. If you pass NULL in this parameter, the Window Manager provides a default menu and sends a Reveal Object Apple event to the Finder if a menu item is selected. Note that in order to pass NULL, there must be a file currently associated with the window. If you pass in a menu handle, this menu supersedes the default window path pop-up menu, and the WindowPathSelect function temporarily inserts the specified menu into the current pop-up menu list. There does not have to be a file currently associated with the window if you pass in your own menu handle.
outMenuResult	A pointer to a value that, on return, contains the menu and menu item the user chose. The high-order word of the value produced contains the menu ID, and the low-order word contains the item number of the menu item. If the user does not select a menu item, 0 is produced in the high-order word, and the low-order word is undefined. For file menus that have not been overridden, 0 is always produced in this parameter. Pass NULL in this parameter if you do not want this information.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

Your application should call the WindowPathSelect function when it detects a Command-click in the title of a window, that is, when the function IsWindowPathSelectClick (page 115) returns a value of true. Calling WindowPathSelect causes the Window Manager to display a window path pop-up menu for your window. Listing 2-5 in "Displaying a Window Path Pop-Up Menu" (page 46) shows an example of how your application might call the WindowPathSelect function.

SPECIAL CONSIDERATIONS

Note that when WindowPathSelect returns noErr, your application should ensure that the window opened by the Finder's Reveal Object Apple event handler is visible to the user. To do this, your application should call the Process Manager function SetFrontProcess with the Finder's process serial number, as shown in Listing 2-6 in "Displaying a Window Path Pop-Up Menu" (page 46).

VERSION NOTES

Available with Mac OS 8.5 and later.

Associating Data With Windows

The Mac OS 8.5 Window Manager provides the following functions for associating data with windows:

- `SetWindowProperty` (page 121) associates an arbitrary piece of data with a window.
- `GetWindowProperty` (page 118) obtains a piece of data that is associated with a window.
- `GetWindowPropertySize` (page 119) obtains the size of a piece of data that is associated with a window.
- `RemoveWindowProperty` (page 120) removes a piece of data that is associated with a window.

GetWindowProperty

Obtains a piece of data that is associated with a window.

```
pascal OSStatus GetWindowProperty (
                    WindowPtr window,
                    PropertyCreator propertyCreator,
                    PropertyTag propertyTag,
                    UInt32 bufferSize,
                    UInt32 *actualSize,
                    void *propertyBuffer);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window to be examined for associated data.

`propertyCreator`
A four-character code. Pass the creator code (typically, the application's signature) of the associated data to be obtained.

`propertyTag` A four-character code. Pass the application-defined code identifying the associated data to be obtained.

`bufferSize`	Pass a value specifying the size of the associated data to be obtained. If the size of the data is unknown, use the function `GetWindowPropertySize` (page 119) to get the data's size. If the size specified does not match the actual size of the property, `GetWindowProperty` only retrieves data up to the size specified or up to the actual size of the property, whichever is smaller, and an error is returned.
`actualSize`	A pointer to a value. On return, the value specifies the actual size of the obtained data. You may pass `NULL` for the `actualSize` parameter if you are not interested in this information.
`propertyBuffer`	A pointer to a buffer. On return, this buffer contains a copy of the data that is associated with the specified window.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

The data retrieved by the `GetWindowProperty` function must have been previously associated with the window with the function `SetWindowProperty` (page 121).

VERSION NOTES

Available with Mac OS 8.5 and later.

GetWindowPropertySize

Obtains the size of a piece of data that is associated with a window.

```
pascal OSStatus GetWindowPropertySize (
                    WindowPtr window,
                    PropertyCreator creator,
                    PropertyTag tag,
                    UInt32 *size);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window to be examined for associated data.

`creator`	A four-character code. Pass the creator code (typically, the application's signature) of the associated data whose size is to be obtained.
`tag`	A four-character code. Pass the application-defined code identifying the associated data whose size is to be obtained.
`size`	A pointer to a value that, on return, specifies the size of the associated data.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

If you want to retrieve a piece of associated data with the `GetWindowProperty` (page 118) function, you typically need to use the `GetWindowPropertySize` function to determine the size of the data beforehand.

VERSION NOTES

Available with Mac OS 8.5 and later.

RemoveWindowProperty

Removes a piece of data that is associated with a window.

```
pascal OSStatus RemoveWindowProperty (
                    WindowPtr window,
                    PropertyCreator propertyCreator,
                    PropertyTag propertyTag);
```

`window`	A value of type `WindowPtr`. Pass a pointer to the window whose data is to be removed.
`propertyCreator`	A four-character code. Pass the creator code (typically, the application's signature) of the associated data to be removed.
`propertyTag`	A four-character code. Pass the application-defined code identifying the associated data to be removed.
function result	A result code. See "Result Codes" (page 148).

DISCUSSION

The data removed by the `RemoveWindowProperty` function must have been previously associated with the window with the function `SetWindowProperty` (page 121).

VERSION NOTES

Available with Mac OS 8.5 and later.

SetWindowProperty

Associates an arbitrary piece of data with a window.

```
pascal OSStatus SetWindowProperty (
                    WindowPtr window,
                    PropertyCreator propertyCreator,
                    PropertyTag propertyTag,
                    UInt32 propertySize,
                    void *propertyBuffer);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window with which data is to be associated.

`propertyCreator`
A four-character code. Pass the creator code (typically, the application's signature) of the data to be associated.

`propertyTag` A four-character code. Pass a value identifying the data to be associated. You define the tag your application uses to identify the data; this code is not to be confused with the file type for the data.

`propertySize` Pass a value specifying the size of the data to be associated.

`propertyBuffer` Pass a pointer to the data to be associated.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

Data set with the `SetWindowProperty` function may be obtained with the function `GetWindowProperty` (page 118) and removed with the function

`RemoveWindowProperty` (page 120). See "Managing Multiple Windows" (page 31) for a discussion of using the `SetWindowProperty` function.

VERSION NOTES

Available with Mac OS 8.5 and later.

Maintaining the Update Region

The Mac OS 8.5 Window Manager provides the following functions for updating windows:

- `InvalWindowRect` (page 122) adds a rectangle to a window's update region.
- `ValidWindowRect` (page 124) removes a rectangle from a window's update region.
- `InvalWindowRgn` (page 123) adds a region to a window's update region.
- `ValidWindowRgn` (page 125) removes a region from a window's update region.

InvalWindowRect

Adds a rectangle to a window's update region.

```
pascal OSStatus InvalWindowRect (
                    WindowPtr window,
                    const Rect *bounds);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window containing the rectangle that you wish to be updated.

`bounds` A pointer to a structure of type `Rect`. Before calling `InvalWindowRect`, set this structure to specify, in local coordinates, a rectangle to be added to the window's update region.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `InvalWindowRect` function informs the Window Manager that an area of a window should be redrawn. The `InvalWindowRect` function is similar to the `InvalRect` function, but `InvalWindowRect` allows the window that it operates upon to be explicitly specified, instead of operating on the current graphics port, so `InvalWindowRect` does not require the graphics port to be set before its use. See "Maintaining the Update Region" (page 52) for further discussion.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `ValidWindowRect` (page 124).

The function `InvalWindowRgn` (page 123).

InvalWindowRgn

Adds a region to a window's update region.

```
pascal OSStatus InvalWindowRgn (
                    WindowPtr window,
                    RgnHandle region);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window containing the region that you wish to be updated.

`region` A value of type `RgnHandle`. Before calling `InvalWindowRgn`, set this region to specify, in local coordinates, the area to be added to the window's update region.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `InvalWindowRgn` function informs the Window Manager that an area of a window should be redrawn. The `InvalWindowRgn` function is similar to the `InvalRgn` function, but `InvalWindowRgn` allows the window that it operates upon to be explicitly specified, instead of operating on the current graphics port, so

`InvalWindowRgn` does not require the graphics port to be set before its use. See "Maintaining the Update Region" (page 52) for further discussion.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `InvalWindowRect` (page 122).

The function `ValidWindowRgn` (page 125).

ValidWindowRect

Removes a rectangle from a window's update region.

```
pascal OSStatus ValidWindowRect (
                    WindowPtr window,
                    const Rect *bounds);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window containing the rectangle that you wish to remove from being updated.

`bounds` A pointer to a structure of type `Rect`. Before calling `ValidWindowRect`, set this structure to specify, in local coordinates, a rectangle to be removed from the window's update region.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `ValidWindowRect` function informs the Window Manager that an area of a window no longer needs to be redrawn. The `ValidWindowRect` function is similar to the `ValidRect` function, but `ValidWindowRect` allows the window that it operates upon to be explicitly specified, instead of operating on the current graphics port, so `ValidWindowRect` does not require the graphics port to be set before its use. See "Maintaining the Update Region" (page 52) for further discussion.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `InvalWindowRect` (page 122).

The function `ValidWindowRgn` (page 125).

ValidWindowRgn

Removes a region from a window's update region.

```
pascal OSStatus ValidWindowRgn (
                    WindowPtr window,
                    RgnHandle region);
```

`window` A value of type `WindowPtr`. Pass a pointer to the window containing the region that you wish to remove from being updated.

`region` A value of type `RgnHandle`. Before calling `ValidWindowRgn`, set this region to specify, in local coordinates, the area to be removed from the window's update region.

function result A result code. See "Result Codes" (page 148).

DISCUSSION

The `ValidWindowRgn` function informs the Window Manager that an area of a window no longer needs to be redrawn. The `ValidWindowRgn` function is similar to the `ValidRgn` function, but `ValidWindowRgn` allows the window that it operates upon to be explicitly specified, instead of operating on the current graphics port, so `ValidWindowRgn` does not require the graphics port to be set before its use. See "Maintaining the Update Region" (page 52) for further discussion.

VERSION NOTES

Available with Mac OS 8.5 and later.

SEE ALSO

The function `InvalWindowRgn` (page 123).

The function `ValidWindowRect` (page 124).

Data Types

The following data types are available with the Mac OS 8.5 Window Manager:

- `BasicWindowDescription` (page 126)
- `MeasureWindowTitleRec` (page 128)
- `SetupWindowProxyDragImageRec` (page 129)

BasicWindowDescription

The `BasicWindowDescription` structure is a default collection item for a resource of type `'wind'` (page 130). You use the `BasicWindowDescription` structure to describe the statically-sized base characteristics of a window.

```
struct BasicWindowDescription {
    UInt32                  descriptionSize;
    Rect                    windowContentRect;
    Rect                    windowZoomRect;
    UInt32                  windowRefCon;
    UInt32                  windowStateFlags;
    WindowPositionMethod    windowPositionMethod;
    UInt32                  windowDefinitionVersion;

    union {

        struct {
            SInt16              windowDefProc;
            Boolean             windowHasCloseBox;
        } versionOne;

        struct {
            WindowClass         windowClass;
```

```
            WindowAttributes    windowAttributes;
        } versionTwo;

    } windowDefinition;

};
typedef struct BasicWindowDescription BasicWindowDescription;
```

Field descriptions

`descriptionSize` A value specifying the size of the entire `BasicWindowDescription` structure.

`windowContentRect` A structure of type `Rect`, specifying the initial size and screen location of the window's content area.

`windowZoomRect` Reserved.

`windowRefCon` The window's reference value field, which is simply storage space available to your application for any purpose. The value contained in this field persists when the `'wind'` resource is stored, so you should avoid saving pointers in this field, as they may become stale.

`windowStateFlags` A 32-bit value whose bits you set to indicate the status of transient window states. See "BasicWindowDescription State Constant" (page 134) for possible values.

`windowPositionMethod` The specification last used in the function `RepositionWindow` (page 94) to position this window, if any. See "RepositionWindow Constants" (page 136) for a description of possible values for this field.

`windowDefinitionVersion` The version of the window definition used for the window. Set this field to a value of 1 if your application is creating a pre–Mac OS 8.5 window, that is, a window lacking class and attribute information. Set this field to a value of 2 if your application is creating a window using class and attribute information. See "BasicWindowDescription Version Constants" (page 135) for descriptions of these values.

`windowDefinition` A union of the `versionOne` and `versionTwo` structures. Your application must either specify the window's class and attributes, or it must supply a window definition ID and

specify whether or not the window has a close box. See "Window Class Constants" (page 140) and "Window Attribute Constants" (page 138) for descriptions of class and attribute values.

MeasureWindowTitleRec

If you implement a custom window definition function, when the Window Manager passes the message `kWindowMsgMeasureTitle` in your window definition function's `message` parameter it also passes a pointer to a structure of type `MeasureWindowTitleRec` in the `param` parameter. Your window definition function is responsible for setting the contents of the `MeasureWindowTitleRec` structure to contain data describing the ideal title width.

See "Window Definition Message Constants" (page 143) and "Window Definition Feature Constants" (page 141) for more details on the `kWindowMsgMeasureTitle` message and the corresponding `kWindowCanMeasureTitle` feature flag.

```
struct MeasureWindowTitleRec
{
/* output parameters*/
    SInt16      fullTitleWidth;     /* text width + proxy icon width */
    SInt16      titleTextWidth;     /* text width only */
/* input parameters*/
    Boolean     isUnicodeTitle;     /* future use */
    Boolean     reserved;           /* future use */
};
typedef struct MeasureWindowTitleRec MeasureWindowTitleRec;
typedef MeasureWindowTitleRec *MeasureWindowTitleRecPtr;
```

Field descriptions

`fullTitleWidth` Your window definition function sets this field to a value specifying the total width in pixels of the window title text and any proxy icon that may be present, ignoring any compression or truncation that might be required when the title is actually drawn. That is, the specified width should be the ideal width that would be used if the window were sufficiently wide to draw the entire title along with a proxy icon. You should measure the title width using the current

	system font. If no proxy icon is present, this field should have the same value as the `titleTextWidth` field.
`titleTextWidth`	Your window definition function sets this field to a value specifying the width in pixels of the window title text, ignoring any compression or truncation that might be required when the title is actually drawn. That is, the specified width should be the ideal width that would be used if the window were sufficiently wide to draw the entire title. You should measure the title width using the current system font.
`isUnicodeTitle`	Your window definition function may ignore this field; it is reserved for future use.
`reserved`	Your window definition function may ignore this field; it is reserved for future use.

SetupWindowProxyDragImageRec

If you implement a custom window definition function, when the function `TrackWindowProxyDrag` (page 112) is called, the Window Manager passes the message `kWindowMsgSetupProxyDragImage` in your window definition function's `message` parameter and passes a pointer to a structure of type `SetupWindowProxyDragImageRec` in the `param` parameter. Your window definition function is responsible for setting the contents of the `SetupWindowProxyDragImageRec` structure to contain data describing the proxy icon's drag image.

See "Window Definition Message Constants" (page 143) and "Window Definition Feature Constants" (page 141) for more details on the `kWindowMsgSetupProxyDragImage` message and the corresponding `kWindowCanSetupProxyDragImage` feature flag.

```
struct SetupWindowProxyDragImageRec
{
    GWorldPtr imageGWorld;
    RgnHandle imageRgn;
    RgnHandle outlineRgn;
};
typedef struct SetupWindowProxyDragImageRec SetupWindowProxyDragImageRec;
```

Field descriptions

`imageGWorld`	A pointer to the offscreen graphics world containing the drag image. The window definition function must allocate the offscreen graphics world, since the Window Manager has no way of knowing the appropriate size for the drag image. The Window Manager disposes of the offscreen graphics world.
`imageRgn`	A handle to a region containing the drag image. Only this portion of the offscreen graphics world referred to by the `imageGWorld` field is actually drawn. The Window Manager allocates and disposes of this region.
`outlineRgn`	A handle to a region containing an outline of the drag image, for use on monitors incapable of displaying the drag image itself. The Window Manager allocates and disposes of this region.

Resources

The following resource is available with the Mac OS 8.5 Window Manager:

- `'wind'` (page 130)

'wind'

Windows can be stored in flattened collections in extensible window resources of type `'wind'`. You create a window from a `'wind'` resource when you call the function `CreateWindowFromResource` (page 69). For more details on collections, see "Collection Manager" in *Inside Macintosh: QuickDraw GX Environment and Utilities.*

Note that due to the complexity of this format, it is possible to create `'wind'` resources using Rez, but it is not possible to DeRez them. DeRez cannot currently handle multiple undefined labels as used in this type definition.

Note, too, that your application's `'wind'` resources must have resource ID numbers greater than 127.

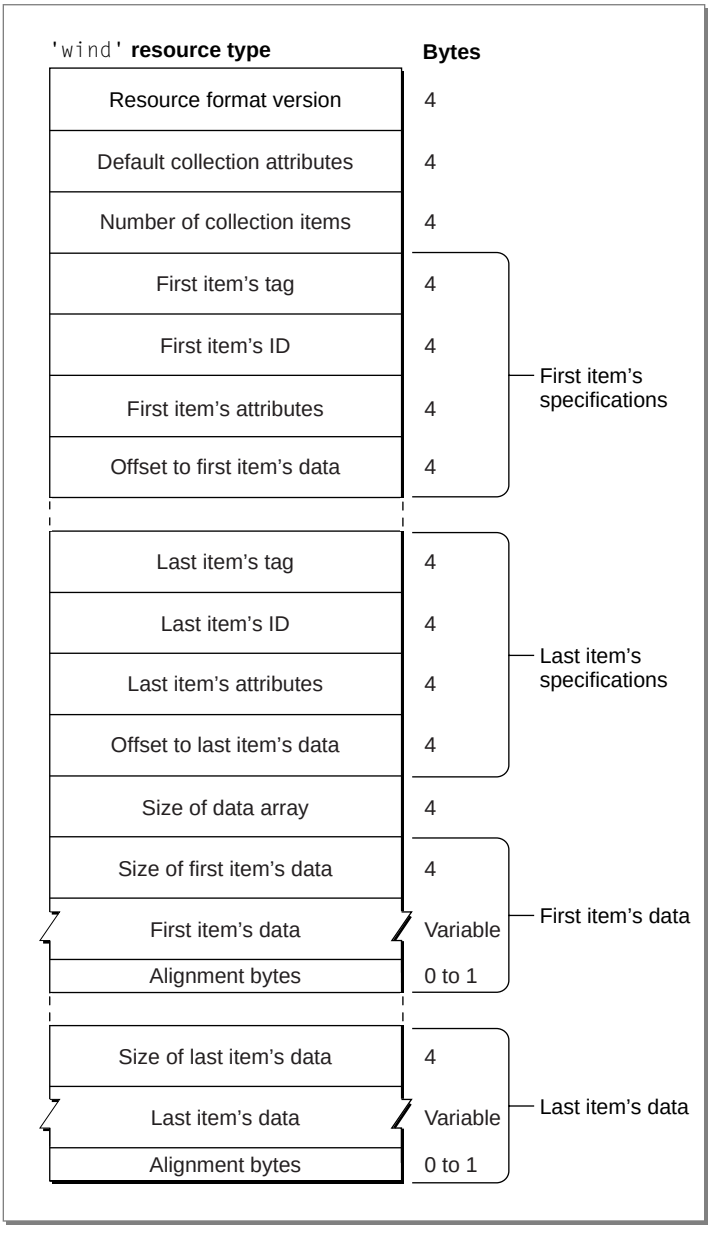
There are currently two default collection items defined for the extended window resource. One default item is a structure of type

`BasicWindowDescription` (page 126), which defines a standard Mac OS 8.5 Window Manager window. The other default item is a Pascal title string for the window. Future versions of the Window Manager may add new default collection items to the format without the application's knowledge.

Application developers are welcome to extend `'wind'` resources with new collection items as they see fit (although zero-length items aren't supported). However, developers may not define new collection items using the `'appl'` collection item tag, which is reserved for use by Apple Computer, Inc. See "'wind' Resource Default Collection Item Constants" (page 138) for details on the tags and the IDs that are reserved for identifying default items.

The format of a compiled `'wind'` resource is based upon that of a `'flac'`, or flattened collection, resource. Figure 3-1 illustrates the format of this resource.

Figure 3-1 Structure of a compiled 'wind' resource



A compiled version of the `'wind'` resource contains the following elements:

- A 32-bit value identifying the version of the `'wind'` resource's format. This value should be set to `0x00010000`.
- Default attribute bits for the collection as a whole; 0 for none.
- The total number of items in the collection.
- An array of items, sorted by tag and ID. Each entry in the item array has corresponding data in the data array. Each item array entry must contain these elements:
  - A tag identifying the item type.
  - An ID identifying the particular item.
  - Thirty-two attributes, each represented by one bit flag, stored in a 32-bit word. The bits are numbered from 0 to 31, with bit 31 being the high bit. The upper 16 bits of an item's attributes are reserved for use by Apple Computer, Inc. The lower 16 bits are attributes that you can define for purposes suitable to your application. Currently, two of the reserved attributes are defined:

    Bit 31 — The lock attribute. When an item has this attribute set, attempts to replace the item result in an error.

    Bit 30 — The persistence attribute. When an item has this attribute set, the item is included when the collection is flattened.

  - An offset to the item's data.
- A value representing the total size of all the items' data in the collection's data array.
- An array of data, corresponding to the array of items. Each entry in the data array must be in the same order as its corresponding entry in the item array. Each data array entry must contain these elements:
  - A value representing the size of the data for the item.
  - The data for the item, which can be of variable size.
  - Alignment bytes. Zero or one bytes used to make the previous data string end on a word boundary.

Constants

The following constants are available with the Mac OS 8.5 Window Manager:

BasicWindowDescription State Constant

You can use the following mask to set a bit in the `windowStateFlags` field of a structure of type `BasicWindowDescription` (page 126), thereby specifying a transient window state.

```
enum {
    kWindowIsCollapsedState = (1 << 0L)
};
```

Constant description

`kWindowIsCollapsedState`
: If the bit specified by this mask is set, the window is currently collapsed.

BasicWindowDescription Version Constants

You may supply one of the following values in the `windowDefinitionVersion` field of a structure of type `BasicWindowDescription` (page 126) to specify the version of the window definition used for a window.

```
enum {
    kWindowDefinitionVersionOne = 1,
    kWindowDefinitionVersionTwo = 2
};
```

Constant descriptions

`kWindowDefinitionVersionOne`
: Specifies a pre–Mac OS 8.5 Window Manager window. Windows of this version are created using a window definition ID and a Boolean value indicating whether or not the window has a close box.

`kWindowDefinitionVersionTwo`
: Specifies a Mac OS 8.5 Window Manager window. Windows of this version are created using class and attribute information. For details on classes and attributes, see “Window Class Constants” (page 140) and “Window Attribute Constants” (page 138), respectively.

FindWindow Result Code Constant for the Proxy Icon

With the Mac OS 8.5 Window Manager, the `FindWindow` function may return the following constant to identify the cursor location at the time the user pressed the mouse button. See *Mac OS 8 Window Manager Reference* for information on other `FindWindow` result code constants.

```
enum {
    inProxyIcon = 12
};
```

Constant description

`inProxyIcon` The user has pressed the mouse button while the cursor is in the proxy icon of a window. When `FindWindow` returns `inProxyIcon`, your application typically calls the function `TrackWindowProxyDrag` (page 112). See "Tracking a Window Proxy Icon Drag" (page 45) and "Displaying a Window Path Pop-Up Menu" (page 46) for examples of how your application might respond to receiving `inProxyIcon` from `FindWindow`.

RepositionWindow Constants

To specify the factors that determine how a window should be positioned, you supply one of the following `WindowPositionMethod` constants to the function `RepositionWindow` (page 94) or in the `BasicWindowDescription` structure of a resource of type `'wind'` (page 130). Do not confuse the `WindowPositionMethod` constants with the pre–Mac OS 8.5 Window Manager window positioning constants or use the `WindowPositionMethod` constants where the older constants are required (such as in the `StandardAlert` function or in `'WIND'`, `'DLOG'`, or `'ALRT'` resources).

```
enum {
    kWindowCenterOnMainScreen                   = 0x00000001,
    kWindowCenterOnParentWindow                 = 0x00000002,
    kWindowCenterOnParentWindowScreen           = 0x00000003,
    kWindowCascadeOnMainScreen                  = 0x00000004,
    kWindowCascadeOnParentWindow                = 0x00000005,
    kWindowCascadeOnParentWindowScreen          = 0x00000006,
    kWindowAlertPositionOnMainScreen            = 0x00000007,
    kWindowAlertPositionOnParentWindow          = 0x00000008,
    kWindowAlertPositionOnParentWindowScreen    = 0x00000009
};
typedef UInt32 WindowPositionMethod;
```

Constant descriptions

`kWindowCenterOnMainScreen`
Center the window, both horizontally and vertically, on the screen that contains the menu bar.

`kWindowCenterOnParentWindow`
: Center the window, both horizontally and vertically, on the parent window. If the window to be centered is wider than the parent window, its left edge is aligned with the parent window's left edge.

`kWindowCenterOnParentWindowScreen`
: Center the window, both horizontally and vertically, on the screen containing the parent window.

`kWindowCascadeOnMainScreen`
: Place the window just below the menu bar at the left edge of the main screen. Subsequent windows are placed on the screen relative to the first window, such that the frame of the preceding window remains visible behind the current window. The exact amount by which windows are offset depends upon the dimensions of the window frame under a given appearance.

`kWindowCascadeOnParentWindow`
: Place the window a distance below and to the right of the upper-left corner of the parent window such that the frame of the parent window remains visible behind the current window. The exact amount by which windows are offset depends upon the dimensions of the window frame under a given appearance.

`kWindowCascadeOnParentWindowScreen`
: Place the window just below the menu bar at the left edge of the screen containing the parent window. Subsequent windows are placed on the screen relative to the first window, such that the frame of the preceding window remains visible behind the current window. The exact amount by which windows are offset depends upon the dimensions of the window frame under a given appearance.

`kWindowAlertPositionOnMainScreen`
: Center the window horizontally and position it vertically on the screen that contains the menu bar, such that about one-fifth of the screen is above it.

`kWindowAlertPositionOnParentWindow`
: Center the window horizontally and position it vertically such that about one-fifth of the parent window is above it.

`kWindowAlertPositionOnParentWindowScreen`
: Center the window horizontally and position it vertically such that about one-fifth of the screen containing the parent window is above it.

'wind' Resource Default Collection Item Constants

The following constants specify the tag and the IDs that identify the default collection items contained in a resource of type `'wind'` (page 130).

```
enum {
    kStoredWindowSystemTag          = 'appl',
    kStoredBasicWindowDescriptionID = 'sbas',
    kStoredWindowPascalTitleID      = 's255'
};
```

Constant descriptions

`kStoredWindowSystemTag`
: This item tag specifies a system-defined collection item. Note that the `'appl'` collection item tag is reserved for use by Apple Computer, Inc. Do not define new collection items using that tag.

`kStoredBasicWindowDescriptionID`
: In combination with `kStoredWindowSystemTag`, this item ID specifies an item of type `BasicWindowDescription`. See `BasicWindowDescription` (page 126) for details on this type.

`kStoredWindowPascalTitleID`
: In combination with `kStoredWindowSystemTag`, this item ID specifies a Pascal title string.

Window Attribute Constants

The `WindowAttributes` enumeration defines masks your application can use to set or test window feature bits. You can use these masks with the function `CreateNewWindow` (page 67) to set window feature bits, thereby defining a window's attributes. You can also use these masks to test the window feature bits produced by the function `GetWindowAttributes` (page 80), thereby obtaining a window's attributes.

```
enum {
    kWindowNoAttributes                     =   0L,
    kWindowCloseBoxAttribute                =   (1L << 0),
    kWindowHorizontalZoomAttribute          =   (1L << 1),
    kWindowVerticalZoomAttribute            =   (1L << 2),
    kWindowFullZoomAttribute                =   (kWindowVerticalZoomAttribute |
                                                kWindowHorizontalZoomAttribute),
    kWindowCollapseBoxAttribute             =   (1L << 3),
    kWindowResizeableAttribute              =   (1L << 4),
    kWindowSideTitlebarAttribute            =   (1L << 5),
    kWindowNoUpdatesAttribute               =   (1L << 16),
    kWindowNoActivatesAttribute             =   (1L << 17),
    kWindowStandardDocumentAttributes       =   (kWindowCloseBoxAttribute |
                                                kWindowFullZoomAttribute |
                                                kWindowCollapseBoxAttribute |
                                                kWindowResizeableAttribute),
    kWindowStandardFloatingAttributes       =   (kWindowCloseBoxAttribute |
                                                kWindowCollapseBoxAttribute)
};
typedef UInt32 WindowAttributes;
```

Constant descriptions

`kWindowNoAttributes`
: If no bits are set, the window has none of the following attributes.

`kWindowCloseBoxAttribute`
: If the bit specified by this mask is set, the window has a close box.

`kWindowHorizontalZoomAttribute`
: If the bit specified by this mask is set, the window has a horizontal zoom box.

`kWindowVerticalZoomAttribute`
: If the bit specified by this mask is set, the window has a vertical zoom box.

`kWindowFullZoomAttribute`
: If the bits specified by this mask are set, the window has a full—horizontal and vertical—zoom box.

`kWindowCollapseBoxAttribute`
: If the bit specified by this mask is set, the window has a collapse box.

`kWindowResizeableAttribute`
: If the bit specified by this mask is set, the window has a size box.

`kWindowSideTitlebarAttribute`
: If the bit specified by this mask is set, the window has a side title bar. This attribute may be applied only to floating windows, that is, those windows assigned the window class constant `kFloatingWindowClass`. See "Window Class Constants" (page 140) for a description of this constant.

`kWindowNoUpdatesAttribute`
: If the bit specified by this mask is set, the window does not receive update events.

`kWindowNoActivatesAttribute`
: If the bit specified by this mask is set, the window does not receive activate events.

`kWindowStandardDocumentAttributes`
: If the bits specified by this mask are set, the window has the attributes of a standard document window—that is, a close box, full zoom box, collapse box, and size box.

`kWindowStandardFloatingAttributes`
: If the bits specified by this mask are set, the window has the attributes of a standard floating window—that is, a close box and collapse box.

Window Class Constants

The `WindowClass` constants categorize windows into groups of like types. The grouping of windows facilitates the appropriate display (that is, both the look and the front-to-back ordering) and tracking of windows.

You can define a window's class using the function `CreateNewWindow` (page 67) and obtain a window's class using the function `GetWindowClass` (page 81). However, a window's class cannot be altered once the window has been created.

Note that the ordering of the constants in the `WindowClass` enumeration reflects the window classes' relative front-to-back display order.

```
enum {
    kAlertWindowClass           = 1L,
    kMovableAlertWindowClass    = 2L,
    kModalWindowClass           = 3L,
    kMovableModalWindowClass    = 4L,
    kFloatingWindowClass        = 5L,
    kDocumentWindowClass        = 6L
};
typedef UInt32 WindowClass;
```

Constant descriptions

`kAlertWindowClass` Identifies an alert box window.

`kMovableAlertWindowClass`
Identifies a movable alert box window.

`kModalWindowClass` Identifies a modal dialog box window.

`kMovableModalWindowClass`
Identifies a movable modal dialog box window.

`kFloatingWindowClass`
Identifies a window that floats above all document windows. If your application assigns this constant to a window and calls the function `InitFloatingWindows` (page 75), the Window Manager ensures that the window has the proper floating behavior. Supported with Mac OS 8.6 and later.

`kDocumentWindowClass`
Identifies a document window or modeless dialog box window. The Window Manager assigns this class to pre–Mac OS 8.5 Window Manager windows.

Window Definition Feature Constants

With the Mac OS 8.5 Window Manager, your window definition function may report the following new feature flags to reflect the features that your window supports. For descriptions of the messages that correspond to these feature flags, see “Window Definition Message Constants” (page 143). For other window definition feature flags, see “Defining Your Own Window Definition Function” in *Mac OS 8 Window Manager Reference*.

```
enum {
    kWindowSupportsDragHilite           = (1 << 7),
    kWindowSupportsModifiedBit          = (1 << 8),
    kWindowCanDrawInCurrentPort         = (1 << 9),
    kWindowCanSetupProxyDragImage       = (1 << 10).
    kWindowCanMeasureTitle              = (1 << 11),
    kWindowWantsDisposeAtProcessDeath   = (1 << 12)
};
```

Constant descriptions

`kWindowSupportsDragHilite`
: If the bit specified by this mask is set, the window supports the `kWindowMsgDragHilite` message.

`kWindowSupportsModifiedBit`
: If the bit specified by this mask is set, the window supports the `kWindowMsgModified` message.

`kWindowCanDrawInCurrentPort`
: If the bit specified by this mask is set, the window supports the `kWindowMsgDrawInCurrentPort` message.

`kWindowCanSetupProxyDragImage`
: If the bit specified by this mask is set, the window supports the `kWindowMsgSetupProxyDragImage` message.

`kWindowCanMeasureTitle`
: If the bit specified by this mask is set, the window supports the `kWindowMsgMeasureTitle` message.

`kWindowWantsDisposeAtProcessDeath`
: If the bit specified by this mask is set, the window definition function wants to receive a `wDispose` message for the window if it still exists when the application quits. Previously, the Window Manager would send a `wDispose` message only if the application explicitly closed the window with calls to the `CloseWindow` or `DisposeWindow` functions. The Window Manager would delete a window that still existed when the application called `ExitToShell` without notifying the window definition function, as part of the destruction of the process.
Note that if a window has the `kWindowWantsDisposeAtProcessDeath` feature bit set, the Window Manager sends your window definition function a

wDispose message for the window when the application exits for any cause, including if your application crashes. A window might want to set this feature flag if it allocates data when it is initialized that lives outside of the application heap and that is not automatically disposed when the application quits. The wDispose message is sent very early in the termination process, so it is still safe for the window definition function to call the system back (for example, you may wish to do this in order to dispose of any auxiliary data). However, to ensure compatibility and to create the minimum performance impact, the window definition function should try to do as little as possible after receiving a wDispose message sent during the termination process.
For further discussion of the wDispose message, see "Defining Your Own Window Definition Function" in *Mac OS 8 Window Manager Reference*.

Window Definition Hit Test Result Code Constant

With the Mac OS 8.5 Window Manager, your window definition function may return the following constant to report that a mouse-down event occurred in your window's proxy icon. For other window definition hit test result code constants, see "Defining Your Own Window Definition Function" in *Mac OS 8 Window Manager Reference*.

```
enum {
    wInProxyIcon    = 10
};
```

Constant description

wInProxyIcon	The mouse-down event occurred in the proxy icon of a window.

Window Definition Message Constants

With the Mac OS 8.5 Window Manager, the Window Manager may pass one of the following constants in the message parameter of your window definition function to specify the action that your function must perform. For descriptions

of the feature bits that correspond to these messages, see "Window Definition Feature Constants" (page 141). For other window definition message constants, see "Defining Your Own Window Definition Function" in *Mac OS 8 Window Manager Reference*.

```
enum {
    kWindowMsgDragHilite            = 9,
    kWindowMsgModified              = 10,
    kWindowMsgDrawInCurrentPort     = 11,
    kWindowMsgSetupProxyDragImage   = 12,
    kWindowMsgStateChanged          = 13,
    kWindowMsgMeasureTitle          = 14
};
```

Constant descriptions

`kWindowMsgDragHilite`

Redraw the window's structure region to reflect the window's validity as a drag-and-drop destination. The Window Manager passes an accompanying Boolean value in your window definition function's `param` parameter. If the value passed is `true`, this indicates that the window's structure region should be highlighted. If the value passed is `false`, the structure region should be unhighlighted. Your window definition function should return 0 as the function result.

`kWindowMsgModified`

Track the window's modification state. The Window Manager sends this message when the function `SetWindowModified` (page 111) is called. The Window Manager passes an accompanying Boolean value in your window definition function's `param` parameter. If the value passed is `true`, the document contained in the window has been modified. If the value passed is `false`, the document has been saved to disk. You should redraw the window's structure region to reflect the new modification state, if appropriate. For example, system-defined document windows dim the proxy icon to indicate that the document has been modified by the user and cannot be moved at that time. Your window definition function should return 0 as the function result.

`kWindowMsgDrawInCurrentPort`

Draw the window's frame in the current graphics port. Other than restricting drawing to the current port, this message is similar to the pre–Mac OS 8.5 Window Manager window definition message constant `wDraw`. See "Drawing the Window Frame" in the "Defining Your Own Window Definition Function" section of *Mac OS 8 Window Manager Reference* for more details on what to do when passed this message.

`kWindowMsgSetupProxyDragImage`

Create the image of the window's proxy icon that the Drag Manager uses to represent the icon while it is being dragged. When your application calls the function `TrackWindowProxyDrag` (page 112), the Window Manager passes this message in your window definition function's `message` parameter and an accompanying pointer to a structure of type `SetupWindowProxyDragImageRec` (page 129) in the `param` parameter. Your window definition function is responsible for setting the contents of the structure to contain the data describing the proxy icon's drag image. Your window definition function should return 0 as the function result.

`kWindowMsgStateChanged`

Be informed that some aspect of the window's public state has changed. The Window Manager passes this message in your window definition function's `message` parameter and an accompanying flag in the `param` parameter that indicates what part of the window's state has been altered. This message is simply a notification message—no response by the window definition function is required. Your window definition function should return 0 as the function result. The `kWindowMsgStateChanged` message is sent after the window's internal data has been updated, but before any redraw occurs onscreen. A window definition function should not redraw the window frame in response to this message. If it is necessary to redraw the window frame, the Window Manager notifies the window definition function with a `wDraw` message. See "Window Definition State-Changed Constant" (page 146) for descriptions of the

values that the Window Manager can pass to specify the state change that has occurred.

`kWindowMsgMeasureTitle`

Measure and return the ideal title width. The Window Manager passes this message in the window definition function's `message` parameter and an accompanying pointer to a structure of type `MeasureWindowTitleRec` (page 128) in the `param` parameter. Your window definition function is responsible for setting the contents of the structure to contain data describing the title width. You should return 0 as the function result.

Window Definition State-Changed Constant

If you implement a custom window definition function, when the Window Manager passes the `kWindowMsgStateChanged` message in your window definition function's `message` parameter it may also pass a value in the `param` parameter with one or more bits set to indicate what part of the window's state has changed. You may use the following mask to test this value. For a description of the `kWindowMsgStateChanged` message, see "Window Definition Message Constants" (page 143).

```
enum {
    kWindowStateTitleChanged        = (1 << 0)
};
```

Constant description

`kWindowStateTitleChanged`

If the bit specified by this mask is set, the window's title has changed.

Window Region Constant for the Proxy Icon Region

With the Mac OS 8.5 Window Manager, you may pass the following `WindowRegionCode` constant to the function `GetWindowRegion` to obtain a handle to the proxy icon region of a window. See *Mac OS 8 Window Manager Reference* for information on the `GetWindowRegion` function and other `WindowRegionCode` constants.

```
enum {
    kWindowTitleProxyIconRgn    = 8
};
```

Constant description

`kWindowTitleProxyIconRgn`
Specifies the region in the window's title area that contains the proxy icon. The proxy icon region is always located within the window's title text region.

Window Transition Action Constants

You may pass the following `WindowTransitionAction` constants to the function `TransitionWindow` (page 78) to specify the direction of the animation effect that is to be performed for a window.

```
enum {
    kWindowShowTransitionAction = 1,
    kWindowHideTransitionAction = 2
};
typedef UInt32 WindowTransitionAction;
```

Constant descriptions

`kWindowShowTransitionAction`
Specifies that the animation display the window opening, that is, transitioning from a closed to an open state.

`kWindowHideTransitionAction`
Specifies that the animation display the window closing, that is, transitioning from an open to a closed state.

Window Transition Effect Constant

You may pass the following `WindowTransitionEffect` constant to the function `TransitionWindow` (page 78) to specify the type of animation effect that is to be performed for a window.

```
enum {
    kWindowZoomTransitionEffect = 1
};
typedef UInt32 WindowTransitionEffect;
```

Constant description

`kWindowZoomTransitionEffect`
Specifies an animation that displays the window zooming between the open and closed states. The direction of the animation, whether from open to closed, or closed to open, depends upon the `WindowTransitionAction` constant specified in conjunction with the `WindowTransitionEffect` constant; see "Window Transition Action Constants" (page 147) for descriptions of possible values.

Result Codes

The most common result codes that the Mac OS 8.5 Window Manager returns are listed below.

`noErr`	0	No error
`errInvalidWindowPtr`	–5600	Invalid window pointer
`errUnsupportedWindowAttributesForClass`	–5601	Attribute bits are inappropriate for the specified window class
`errWindowDoesNotHaveProxy`	–5602	No proxy attached to window
`errInvalidWindowProperty`	–5603	`'appl'` creator code not allowed
`errWindowPropertyNotFound`	–5604	Specified property does not exist
`errUnrecognizedWindowClass`	–5605	Unknown window class
`errCorruptWindowDescription`	–5606	Incorrect size or version supplied in the `BasicWindowDescription` structure

errUserWantsToDragWindow	–5607	Entire window is being dragged, not proxy icon
errWindowsAlreadyInitialized	–5608	Called InitFloatingWindows twice, or called InitWindows and then InitFloatingWindows
errFloatingWindowsNotInitialized	–5609	Called HideFloatingWindows or ShowFloatingWindows without calling InitFloatingWindows

Document Version History

This document has had the following releases:

Table A-1 *Programming With the Mac OS 8.5 Window Manager* revision history

Version	Notes
Mar. 8, 1999	Initial public release. The following changes were made from the prior (seed draft) version: Changed "Window Manager 2.0" to "Mac OS 8.5 Window Manager" throughout to reflect final versioning. Added "Using the Mac OS 8.5 Window Manager" and "About the Mac OS 8.5 Window Manager" chapters to contain programming discussions, code listings, artwork, and conceptual material. "Gestalt Constants" (page 65). Added description of the `gestaltHasFloatingWindows` bit. `CreateNewWindow` (page 67). Changed function name to `CreateNewWindow` from `CreateWindow` to reflect final naming. `AreFloatingWindowsVisible` (page 73), `HideFloatingWindows` (page 74), and `ShowFloatingWindows` (page 77). Noted requirement for each of these functions that the `InitFloatingWindows` function be called prior to their use and that these functions are therefore not supported under Mac OS 8.5 (or prior system versions). `InitFloatingWindows` (page 75). Added description of this function. `MoveWindowStructure` (page 93), `RepositionWindow` (page 94), and `SetWindowBounds` (page 96). Noted that these functions display the window after changing its size and / or position. `ResizeWindow` (page 95). Noted that this function is not supported under Mac OS 8.5 (or prior Mac OS versions). Corrected description of `sizeConstraints` parameter to note that 32,767 is the largest maximum value that can be passed and that `NULL` may be passed, as well.

Table A-1 *Programming With the Mac OS 8.5 Window Manager* revision history

Version	Notes
	`RemoveWindowProxy` (page 100), `SetWindowProxyAlias` (page 101), `SetWindowProxyCreatorAndType` (page 102), `SetWindowProxyFSSpec` (page 104), and `SetWindowProxyIcon` (page 105). Noted requirement under Mac OS 8.5 to set graphics port before drawing, after calls to these functions.
	`EndWindowProxyDrag` (page 108). Changed parameter name to `theDrag` from `drag` to reflect final naming.
	`HiliteWindowFrameForDrag` (page 110). Added more information on the functions `ShowDragHilite` and `HideDragHilite` to discussion.
	`SetWindowModified` (page 111). Expanded discussion to clarify that the state of the content of the window is what the modification state of the window should reflect.
	`IsWindowPathSelectClick` (page 115). Corrected to discussion to note that `IsWindowPathSelectClick` should be called when `FindWindow` returns either `inDrag` or `inProxyIcon`.
	`WindowPathSelect` (page 116). Noted that your program must ensure that the Finder window resulting from this call is brought to the front.
	`GetWindowProperty` (page 118), `RemoveWindowProperty` (page 120), and `SetWindowProperty` (page 121). Changed parameter names to reflect final naming. Also noted that `NULL` may be passed in the `actualSize` parameter of `GetWindowProperty`.
	`MeasureWindowTitleRec` (page 128). Changed the name of the first reserved field to "`isUnicodeTitle`" to reflect final naming.
	"Window Class Constants" (page 140). Noted that one must call the function `InitFloatingWindows` (page 75) for windows assigned the `kFloatingWindowClass` constant.
	"Window Definition State-Changed Constant" (page 146). Removed unimplemented constant values.
	"Result Codes" (page 148). Added descriptions of the `errWindowsAlreadyInitialized` and `errFloatingWindowsNotInitialized` result codes.
Apr. 2, 1998	First seed draft release.

Index

H

I, J

K, L

M, N, O

P, Q

R

S

T

U

V

W, X, Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Donna S. Lee

ILLUSTRATORS
Ruth Anderson, David Arrigoni

PRODUCTION EDITORS
Glen Frank, Gerri Gray

Acknowledgments to Pete Gontier, Eric Schlegel, Chris Thomas, and Ed Voas.