CHAPTER 4

Window Manager

Contents

This chapter describes how your application can use the Window Manager to create and manage windows.

A Macintosh application uses windows for most communication with the user, from discrete interactions like presenting and acknowledging alert boxes to open-ended interactions like creating and editing documents. Users generally type words and formulas, draw pictures, or otherwise enter data in a window on the screen. Your application typically lets the user save this data in a file, open saved files, and view the saved data in a window. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for more information about handling files.

A window can be any size or shape, and the user can display any number of windows, within the limits of available memory, on the screen at once.

The Window Manager defines a set of standard windows and provides a set of routines for managing them. The Window Manager helps your application display windows that are consistent with the Macintosh user interface. See *Macintosh Human Interface Guidelines* for a detailed description of windows and their behavior.

You typically store information about your windows in resources. This chapter describes the standard window resources. For general information on resources, see the chapter "Introduction to the Macintosh Toolbox" in this book. For information on Resource Manager routines, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

The Window Manager itself depends on QuickDraw, the part of the Macintosh system software that handles quick manipulation of graphics. QuickDraw supports drawing into graphics ports, which are individual and complete drawing environments with independent coordinate systems. Each window represents a graphics port, which is described in *Inside Macintosh: Imaging*.

To maintain its windows, your application needs to know what actions the user is taking on the desktop. It receives this information through events, which are messages that describe user actions and report on the processing status of your application. This chapter describes the events that affect window display and considers mouse-down and keyboard events as they relate to windows. For a complete description of events and how your application handles them, see the chapter "Event Manager" in this book.

Most document windows contain controls, which are screen images the user manipulates to control the display or the behavior of the application. This chapter illustrates the controls most commonly used in windows. For more information on creating and responding to controls, see the chapter "Control Manager" in this book.

You use the Window Manager to create and display a new window when the user creates a new document or opens an existing document. When the user clicks or holds down the mouse button while the cursor is in a window created by your application, you use the Window Manager to determine the location of the mouse action and to alter the window display as appropriate. When the user closes a window, you use the Window Manager to remove the window from the screen.

This chapter describes how the Window Manager supports windows and then explains how you can use the Window Manager to

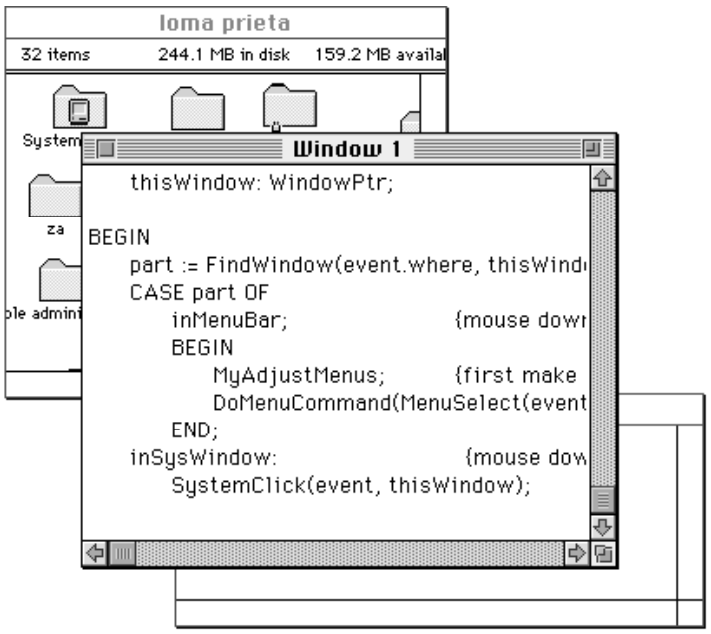
- create and display windows
- handle events in windows
- change the display when the user moves or resizes windows
- remove windows

Introduction to Windows

A **window** is a user interface element, an area on the screen in which the user can enter or view information.

The user can have multiple windows on the desktop at once, from a number of different applications. The user can change the size and location of most windows and can place windows entirely or partially in front of other windows. Figure 4-1 shows a few windows on the desktop.

Figure 4-1 Multiple windows



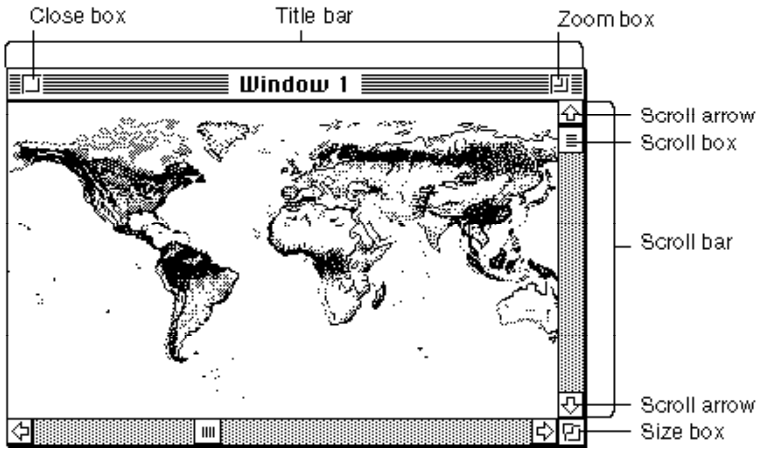
Your application typically creates **document windows** that allow the user to enter and display text, graphics, or other information. For an illustration of a document window in full color, see Plate 1 at the beginning of this book.

A document window is a view into the document—if the document is larger than the window, the window is a view of a portion of the document. Your application can put one or more windows on the screen, each window showing a view of a document or of auxiliary information used to process the document.

The Window Manager defines and supports a set of standard window elements through which the user can manipulate windows. It's important that your application follow the standard conventions for drawing, moving, resizing, and closing windows. By presenting the standard interface, you make experienced users instantly familiar with many aspects of your application, allowing them to focus on learning its unique features.

Figure 4-2 illustrates a standard document window and its elements.

Figure 4-2 A document window



The **title bar** displays the name of the window and indicates whether it's active or not. The Window Manager displays the title of the window in the center of the title bar, in the system font and system font size. If the system font is in the Roman script system, the title bar is 20 pixels high.

When the user creates a new document, you ordinarily display a new document window with the title "untitled", spelled in lowercase letters. If the user creates a second new document window without saving the first, you title the second window "untitled 2", with a space between the word and the number. Continue to add 1 to the number in the title as long as the user continues to create new windows without saving previously numbered, untitled windows.

When the user opens a saved document, you assign the document's filename to the window in which it is displayed.

The user expects to move a window by dragging it by its title bar. You can support moving the window by calling the Window Manager's `DragWindow` procedure, as described in "Moving a Window" on page 4-53.

The **close box** offers the user a quick way to close a window. You can use the `TrackGoAway` function to track mouse activity in the close box and the `CloseWindow` and `DisposeWindow` procedures to close windows. Closing windows is described in "Closing a Window" beginning on page 4-60.

The **zoom box** offers the user a quick way to switch between two different window sizes. You use the `TrackBox` function to track mouse activity in the zoom box and the `ZoomWindow` procedure to zoom windows. Zooming windows is described in "Zooming a Window" beginning on page 4-53.

The **size box** lets the user change the size and dimensions of the window. You use the `GrowWindow` function to track mouse activity in the size box and the `SizeWindow` procedure to resize windows. Sizing windows is described in "Resizing a Window" beginning on page 4-57.

The **scroll bars** let the user see different parts of a document that contains more information than can be displayed at once in the window. Although the Macintosh user interface guidelines specify that you place scroll bars on the right and lower edges of a window that needs them, scroll bars are not part of the window structure. You create and control the scroll bars through the Control Manager, described in the chapter "Control Manager" in this book.

The **content region** is the part of the window in which your application displays the contents of a document, the size box, and the window controls.

The window **frame** is the part of the window drawn automatically by the Window Manager—the title bar, including the close box and zoom box, and the window's outline.

The **structure region** is the entire screen area occupied by a window, including the frame and content region. (See Figure 4-10 on page 4-12.)

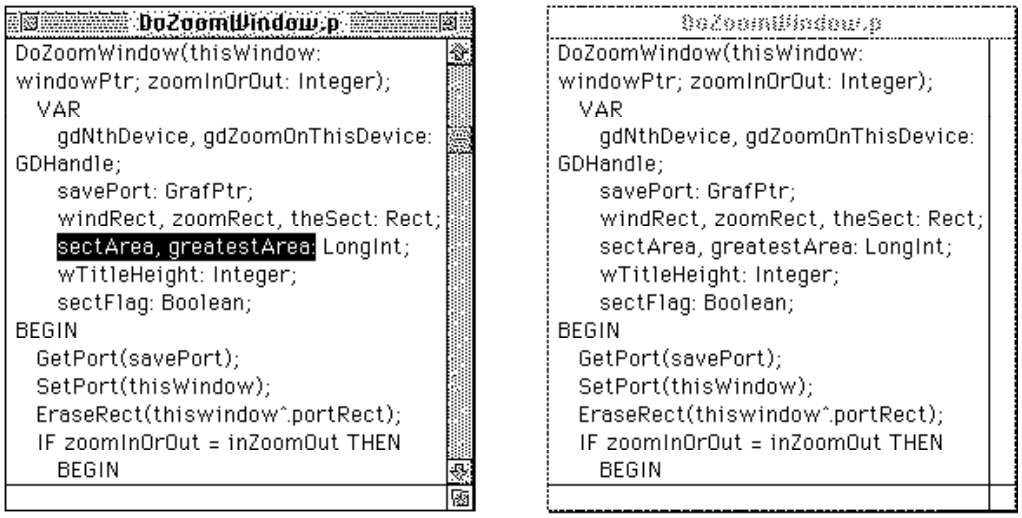
Active and Inactive Windows

The window in which the user is currently working is the **active window.** The active window is the frontmost window on the desktop. It is identified visually by the "racing stripes" in its title bar.

The active window is the target of keyboard activity. It often contains a blinking insertion point (also called the caret) marking the place where new text or graphics will appear. When the user selects text in an active window, your application should highlight the text with inverse video; if the window becomes inactive, you remove the highlighting. You can use a secondary selection technique, such as an outline, to mark a selection in an inactive window. You display scroll bars only in the active window. Figure 4-3 illustrates a sample document window in active and inactive states.

Except for the active window, all document windows on the desktop, whether they belong to your application or another, are inactive. Your application can process documents in inactive windows, but only the active window interacts with the user. For example, if the user chooses Save from the File menu, your application saves only the document in the active window.

Figure 4-3 Active and inactive document windows



Active document window

Inactive document window

To make a window active, the user clicks anywhere in its contents or frame. When the user activates one of your windows, you call the Window Manager to highlight the window frame and title bar; you activate the controls and window contents. As a window becomes active, it appears to the user to move forward, in front of all other windows.

When the user clicks in an inactive document window, you should make the window active but not make any selections in the window in response to the click. To make a selection in the window, the user must click again. This behavior protects the user from losing an existing selection unintentionally when activating a window.

Note

The Finder makes selections in response to the first click in an inactive window, because this action is more natural for the way Finder windows are used. You might find that users expect the first click to cause a selection in some other special-purpose windows created by your application. This behavior is seldom appropriate in document windows. ◆

When a window that belongs to your application becomes inactive, the Window Manager redraws the frame, removing the highlighting from the title bar and hiding the close and zoom boxes. Your application hides the controls and the size box and removes highlighting from application-controlled elements.

When the user reactivates a window, reinstate the window as it was before it was deactivated. Draw the scroll box in the same position and restore the insertion point or highlight the previous selection.

Types of Windows

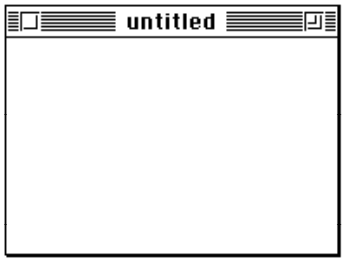
Because windows have so many uses, their appearances vary. The Window Manager defines a number of **window types** that meet the basic needs of most applications. A window type is the general description of how a window looks and behaves. Some windows have title bars and others don't, for example, and windows can have almost any combination of the window-manipulation elements: close box, zoom box, and size box.

This section describes the nine basic window types supported by the Window Manager and their uses. You can create windows of these types by specifying one of the window type constants: `zoomDocProc`, `dBoxProc`, `altDBoxProc`, `plainDBoxProc`, `movableDBoxProc`, `noGrowDocProc`, `documentProc`, `zoomNoGrow`, and `rDocProc`. For instructions for creating windows, see "Creating a Window" beginning on page 4-25.

To give the user maximum flexibility and control, you can use the `zoomDocProc` window type for your document windows. A `zoomDocProc` window supports all of the window-manipulation elements shown in Figure 4-2 on page 4-5: title bar, close box, zoom box, and size box. The Window Manager does not necessarily draw the close box and size box, however. You must call the Window Manager's `DrawGrowIcon` procedure to draw the size box, and you can optionally suppress the close box when you create the window. For more information on defining a window's characteristics, see "Creating a Window" beginning on page 4-25.

Figure 4-4 illustrates a window of type `zoomDocProc` with a close box, as drawn by the Window Manager before you add the size box and scroll bars.

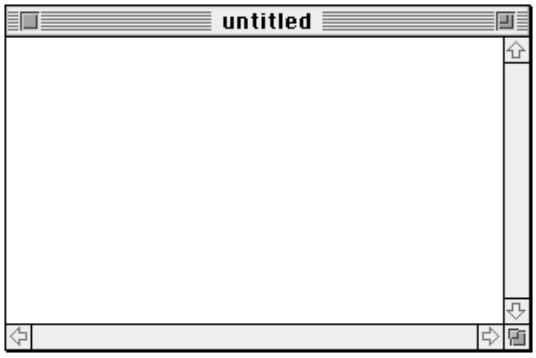
Figure 4-4 A window of type `zoomDocProc`



zoomDocProc

In most cases, a window of type `zoomDocProc` should contain both a close box and a size box. When the related document contains more data than fits in the window, you activate the scroll bars and adjust them to show where in the document the user is working. Figure 4-5 illustrates a window of type `zoomDocProc` with a size box and scroll bars.

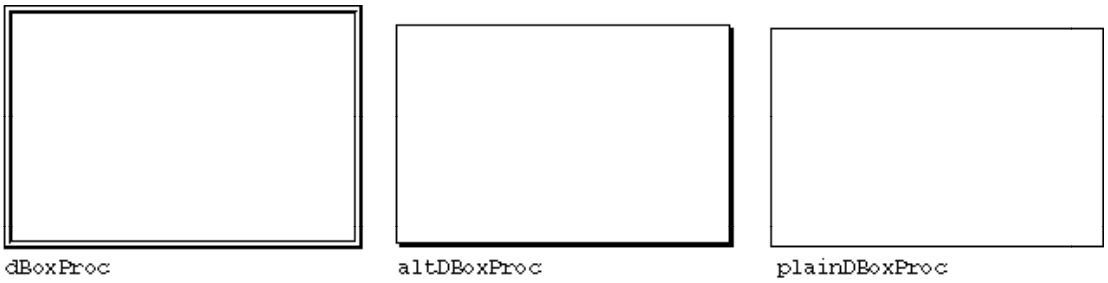
Figure 4-5 A window of type `zoomDocProc`, with size box and inactive scroll bars



You also use windows to display alert boxes and dialog boxes. This section describes the window types used for alert boxes and dialog boxes. For more thorough descriptions of the different kinds of alert boxes and dialog boxes, see the chapter "Dialog Manager" in this book.

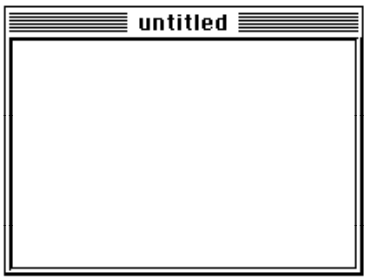
Alert boxes and fixed-position modal dialog boxes contain no window-manipulation elements. The user cannot move, resize, zoom, or close them manually. An alert box or a modal dialog box remains on the screen as the active window until the Dialog Manager or your application removes it—usually when the user completes the interaction by clicking one of the buttons. Figure 4-6 illustrates the three window types available for alert boxes and fixed-position modal dialog boxes.

Figure 4-6 Window types for alert boxes and fixed-position modal dialog boxes



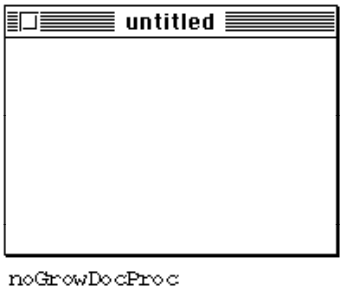
When you want to let the user move a modal dialog box window—in order, for example, to see text that might be obscured by the window—you can implement a movable modal dialog box. A movable modal dialog box cannot be resized, closed, or zoomed, but it can be moved. Figure 4-7 on the next page illustrates the `movableDBoxProc` window type. Like a fixed-position modal dialog box, the movable modal dialog box remains active until the user completes the dialog.

Figure 4-7 A window of type `movableDBoxProc`



Whenever possible, avoid modal dialog boxes and instead use modeless dialog boxes, which allow the user to perform other tasks without dismissing the dialog box. Windows of type `noGrowDocProc`, used for displaying modeless dialog boxes, can be moved or closed but not resized or zoomed. You can implement modeless dialog boxes with other window types if necessary, but it's easier to conform to the user interface guidelines if you keep your dialog box windows as simple as possible. Figure 4-8 illustrates the modeless dialog box window.

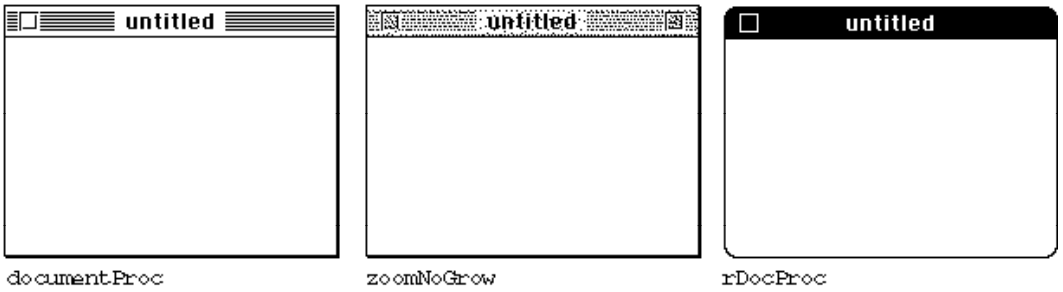
Figure 4-8 A window of type `noGrowDocProc`



The Window Manager also supports a few window types that are seldom used. The `documentProc` window type, for example, has a title bar and supports a close box and size box but no zoom box. The `zoomNoGrow` window type is virtually never appropriate: `zoomNoGrow` supports a close box and a zoom box, but not a size box. The `rDocProc` window type is a rounded-corner window with a title bar and a close box; it is used by desk accessories. Figure 4-9 illustrates these three seldom-used window types.

The **window definition function** defines the general appearance and behavior of a window. The system software and various Window Manager routines call a window's window definition function when they need to perform certain window-dependent actions, such as drawing or resizing a window's frame.

Figure 4-9 Seldom-used window types



The Window Manager supplies two standard window definition functions that handle the nine standard window types. A window definition function draws the window's frame, draws the close box and window title (if any), determines which region the cursor is in within the window, calculates the window's structure and content regions, draws the window's zoom box (if any), draws the window's size box (if any), and performs any special initialization or disposal tasks.

A single window definition function can support up to 16 different window types. The window definition function defines a **variation code,** an integer from 0 through 15, for each window type it supports.

A **window definition ID** is a single value incorporating both the window's definition function and its variation code. (The resource ID of the window definition function is stored in the upper 12 bits of the integer, and the variation code is stored in the lower 4 bits.) The window-type constants described in this section are in fact window definition IDs.

Constant	Window definition ID	Description
`documentProc`	0	movable, sizable window, no zoom box
`dBoxProc`	1	alert box or modal dialog box
`plainDBox`	2	plain box
`altDBoxProc`	3	plain box with shadow
`noGrowDocProc`	4	movable window, no size box or zoom box
`movableDBoxProc`	5	movable modal dialog box
`zoomDocProc`	8	standard document window
`zoomNoGrow`	12	zoomable, nonresizable window
`rDocProc`	16	rounded-corner window

You can provide your own window definition function if you need a window with unusual characteristics, as described in "The Window Definition Function" beginning on page 4-120. Always be careful to conform window behavior to the guidelines in *Macintosh Human Interface Guidelines.*

Window Regions

The Window Manager recognizes a number of different special-purpose **window regions,** which are defined by either the Window Manager or the window definition functions.

The most obvious window regions are the parts of the visible window that the user manipulates to control the display. These window regions correspond to the standard window parts. The **drag region** is the area occupied by the title bar, except for the close box and zoom box. (The user moves the window by dragging it by its title bar.) The **size region, close region,** and **zoom region** are the areas occupied by the size box, close box, and zoom box, respectively.

When the user presses the mouse button while the cursor is in one of your windows, you use the Window Manager function `FindWindow` to determine the region in which the mouse-down event occurred. (The `FindWindow` function calls the window's window definition function, which defines and interprets the window-manipulation regions.) Depending on the result, you then call the appropriate Window Manager routine or your own routine for handling the event. For more information about determining where the cursor is when the user presses the mouse button, see "Handling Mouse Events in Windows" on page 4-42. For discussions of how to use the Window Manager routines for moving, sizing, closing, and zooming windows, see "Moving a Window" beginning on page 4-53 and the sections that follow it.

The Window Manager also makes a broad distinction between the parts of the window it draws automatically and the parts drawn by your application. The Window Manager draws the window frame—the title bar, including the close box and zoom box, and the window's outline. (The Window Manager also draws the size box, but only when your application calls the `DrawGrowIcon` procedure.) Your application is responsible for drawing the content region—that is, the part of the window in which the contents of a document, the size box, and the window controls (including the scroll bars) are displayed.

The entire screen area occupied by a window, including the window outline, title bar, and content region, is the structure region. Figure 4-10 illustrates the frame, content region, and structure region of a window.

Figure 4-10 Window frame, content region, and structure region

The drawing region of a graphics port associated with a window encompasses only the window's content region.

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 redraws the update region. Both your application and the Window Manager can manipulate a window's update region. The sections "Updating the Content Region" on page 4-40 and "Maintaining the Update Region" on page 4-41 describe how the Window Manager and your application track and use the update region.

Dialog Boxes and Alert Boxes

Macintosh applications use alert boxes and dialog boxes to give the user messages and to solicit information. A text-processing application, for example, might display an alert box telling the user that a newly inserted graphic does not fit within the page boundaries. It might display a dialog box in which the user can specify margins, tabs, and other formatting information. (The chapter "Dialog Manager" in this book explains how to use the various kinds of alert boxes and dialog boxes.)

Alert boxes and dialog boxes are merely special-purpose windows. You can handle all alert boxes and most modal dialog boxes through the Dialog Manager, which itself calls the Window Manager. You supply the Dialog Manager with lists of the items in your alert boxes and dialog boxes, and the Dialog Manager displays the windows, tells you which items the user is manipulating, and disposes of the windows when the user is done. Your application provides the code that responds to the user's selections in the alert and dialog boxes.

Although you can specify any window type for your alert boxes and modal dialog boxes, the Dialog Manager functions that handle alert boxes and modal dialog boxes do not support window manipulation. You should therefore use one of the window types without a title bar or size box, most typically the `dBoxProc` window type, for alert boxes and modal dialog boxes. (When the user is responding to a modal dialog box, mouse-down events outside the menu bar or the content region of the dialog box result only in the sounding of the system alert. Note that the Process Manager does not perform major switching while the `ModalDialog` procedure is handling events.)

You use the `movableDBox` window type for movable modal dialog boxes. As described in the chapter "Dialog Manager" in this book, your application can use the Dialog Manager to help handle events in a movable modal dialog box. Your application, however, must handle window-manipulation events—ordinarily only the moving of the movable modal dialog box window.

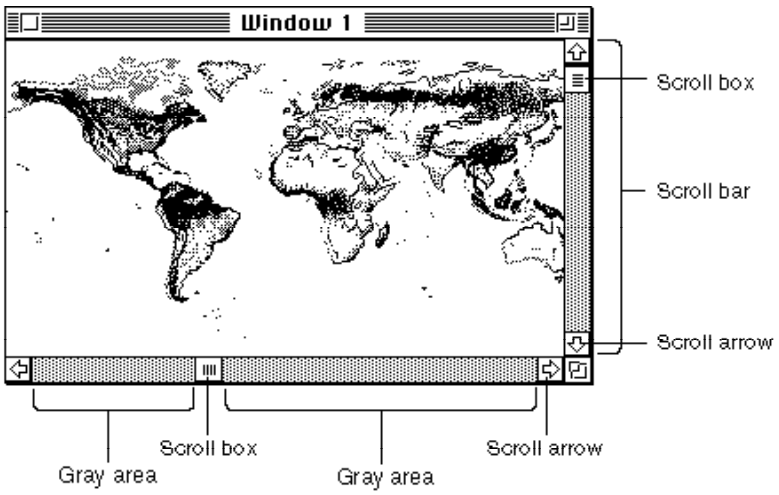
Use the `noGrowDocProc` window type for modeless dialog boxes. You typically use the Dialog Manager to handle events in a modeless dialog box, much like events in a movable modal dialog box. Your application handles window-manipulation events in modeless dialog boxes just as it handles them in document windows.

If you use complex dialog boxes, you might find it's more efficient to use the Window Manager and other parts of the Toolbox, instead of the Dialog Manager, to create and manage your own dialog box windows. Again, see the chapter "Dialog Manager" in this book for a list of characteristics to consider when evaluating the complexity of a dialog box and for examples of customized dialog boxes.

Controls

Most windows contain **controls,** which are screen images that the user manipulates to control the display or the behavior of the application. The most common control in a document window is the scroll bar, illustrated in Figure 4-11.

Figure 4-11 Scroll bars

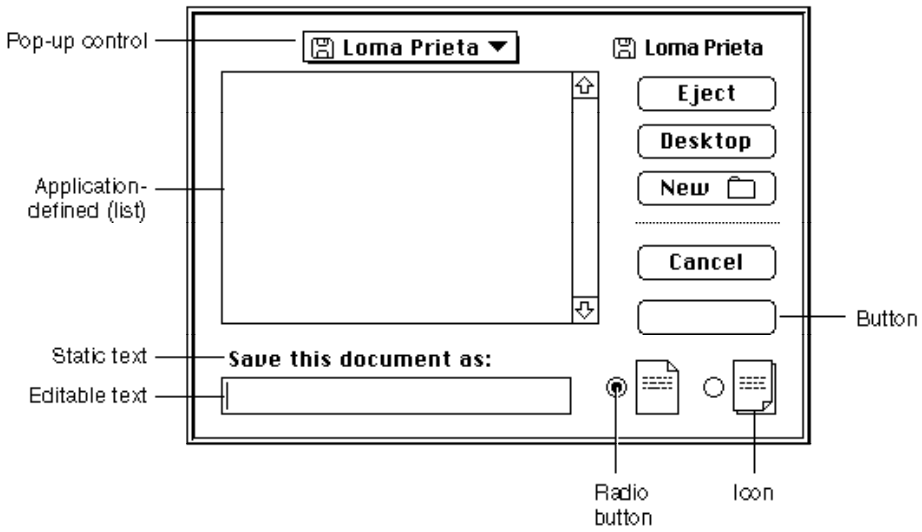


You use scroll bars to show the relative position, within the entire document, of the portion of the document displayed in the window. You should allow the user to drag the scroll box or click in the gray areas or the scroll arrows to move parts of the document into and out of the window. You activate scroll bars in a window any time there is more data than can be shown at one time in the space available.

You use the Control Manager to create, display, and manipulate the scroll bars and any other controls in your windows. Each control "belongs" to a window and is displayed within the graphics port that represents that window. For each window your application creates, the Window Manager maintains a **control list,** a series of entries pointing to the descriptions of the controls associated with the window.

Most alert boxes and dialog boxes contain **buttons,** rounded rectangles that cause an immediate or continuous action when clicked, and most dialog boxes contain additional screen images, like **radio buttons,** that display and retain settings. Figure 4-12 illustrates a dialog box with buttons, radio buttons, and a number of other controls and dialog items.

Figure 4-12 Controls in a dialog box



Buttons ordinarily appear only in alert boxes and dialog boxes. Most of the other elements illustrated in Figure 4-12 appear only in dialog boxes. If you use the Dialog Manager to create your alert boxes and dialog boxes, it draws your controls for you and lets you know when the user has clicked one of them. You can, however, call the Control Manager yourself to display and track buttons and other controls in any windows your application creates. You can also write your own control definition functions to create and control other kinds of controls. For a complete description of how to create and support controls, see the chapter "Control Manager" in this book.

Windows on the Desktop

Multiple windows, from different applications, can appear simultaneously on the desktop. The Window Manager tracks all windows, using its own private data structure called the **window list.** Entries appear in the window list in their order on the desktop, beginning with the frontmost, active window. When the user changes the ordering of windows on the desktop, the Window Manager generates events telling your application to activate, deactivate, and redraw windows as necessary. The Window Manager prevents you from drawing accidentally in the windows of other applications.

The user can interact with only one application at a time. The application with which the user is interacting (that is, the application that owns the window in which the user is working) is the active application, or **foreground process,** and the others are inactive applications, or **background processes.** One way the user can switch applications is by clicking in a window that belongs to a background process. The Process Manager then generates events telling the previously active application that it's about to be suspended and telling the newly active application that it can resume processing. (For more information about the workings of foreground and background processes and about the events that support simultaneous running of multiple applications, see the chapter "Event Manager" in this book.)

Your application is likely to have multiple windows on the desktop at once: one or more document windows, possibly one or more dialog box windows, and possibly some other special-purpose windows. The section "Managing Multiple Windows" beginning on page 4-23 suggests a technique for keeping track of multiple windows.

On the original Macintosh computer, the desktop area was limited to a single screen of known dimensions. Contemporary systems, however, can support multiple monitors of various sizes and capabilities. To place its windows in the appropriate place on the desktop, your application must pay attention to what screen space is available and where the user is working. For the rules governing window placement, see *Macintosh Human Interface Guidelines.* For techniques for managing windows on multiple screens, see "Positioning a Document Window on the Desktop" beginning on page 4-30.

The entire area of the desktop—that is, the screen area that is not occupied by the menu bar—is known as the **gray region.** The Window Manager maintains a pointer to the gray region in a global variable named `GrayRgn`; you can retrieve a pointer to the gray region with the Window Manager function `GetGrayRgn`.

About the Window Manager

The Window Manager provides a complete set of routines for creating, moving, resizing, and otherwise manipulating windows. It also provides lower-level support by managing the layering of windows on the desktop and by alerting your application to desktop changes that affect its windows. Your application and the Window Manager work together to provide the user with a consistent window interface.

When, for example, the user presses the mouse button while the cursor is in the drag region of a window's title bar, you can call the `DragWindow` procedure, which moves a dotted outline of the window around the screen in response to mouse movements. When the user releases the mouse button, `DragWindow` calls the `MoveWindow` procedure, which redraws the window in its new location. If part or all of an inactive window belonging to your application is exposed by the move, the Window Manager triggers an update event that tells your application to redraw the exposed region.

Similarly, if the user clicks in an inactive window, you can call the `SelectWindow` procedure. `SelectWindow` adjusts the window highlighting and layering and

also generates activate events that tell your application which windows to activate and deactivate.

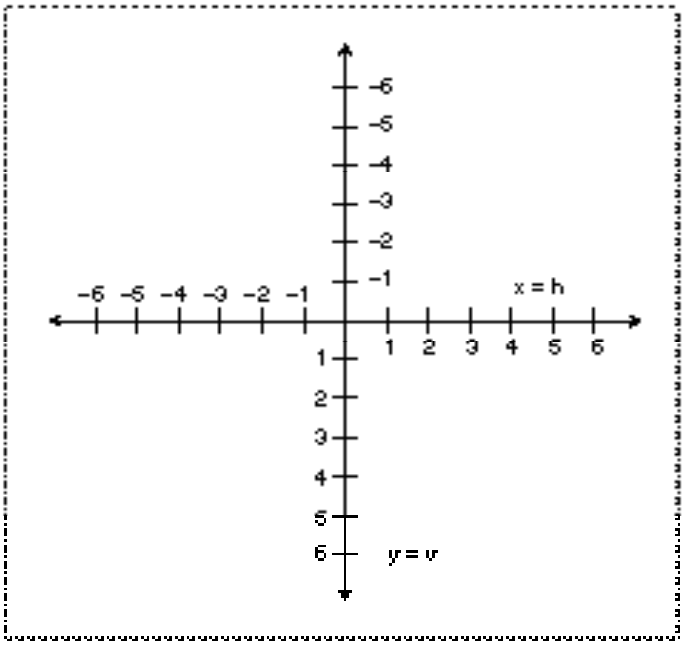
The Window Manager has built-in support for the nine basic window types described in "Types of Windows" beginning on page 4-8. When you are using one of these window types, the Window Manager draws the window's frame, determines what region of the window the cursor is in, calculates the window's structure and content regions, draws the window's size box, draws the window's close box and zoom box, and performs any special initialization or disposal tasks. If necessary, you can write your own window definition function to handle other types of windows.

Graphics Ports

Each window represents a QuickDraw **graphics port,** which is a drawing environment with its own coordinate system. (See *Inside Macintosh: Imaging* for a complete description of graphics ports and coordinate systems.) When you create a window, the Window Manager creates a graphics port in which the window's contents are displayed.

The location of a window on the screen is defined in **global coordinates**—that is, coordinates that reflect the entire potential drawing space. QuickDraw and Color QuickDraw recognize a coordinate plane whose origin is the upper-left corner of the main screen, whose positive x-axis extends rightward, and whose positive y-axis extends downward. In QuickDraw, the horizontal offset is ordinarily labeled h, and the vertical offset v. Figure 4-13 illustrates the QuickDraw global coordinate system.

Figure 4-13 The QuickDraw global coordinate plane



Note
The orientation of the vertical axis, while convenient for computer graphics, differs from mathematical convention. Also, the coordinate plane is bounded by the limits of QuickDraw coordinates, which range from –32,768 to 32,767.

QuickDraw stores points and rectangles in its own data structures of type `Point` and `Rect`. In these structures, the vertical coordinate (`v`) appears first, followed by the horizontal coordinate (`h`). Most, but not all, QuickDraw routines that handle points require you to specify the coordinates in this order. ◆

When QuickDraw creates a new graphics port (usually because you've created a new window through the Window Manager), it defines a bounding rectangle for the port, in global coordinates. Ordinarily, the bounding rectangle represents the entire area of the screen on which the window appears. The bounding rectangle is stored in the graphics port data structure, in the `bounds` field of a structure called a pixel map in Color QuickDraw and a bitmap in QuickDraw.

The graphics port data structure also includes a field called `portRect`, which defines a rectangle to be used for drawing. In a graphics port that represents a window, the `portRect` rectangle represents the window's content region.

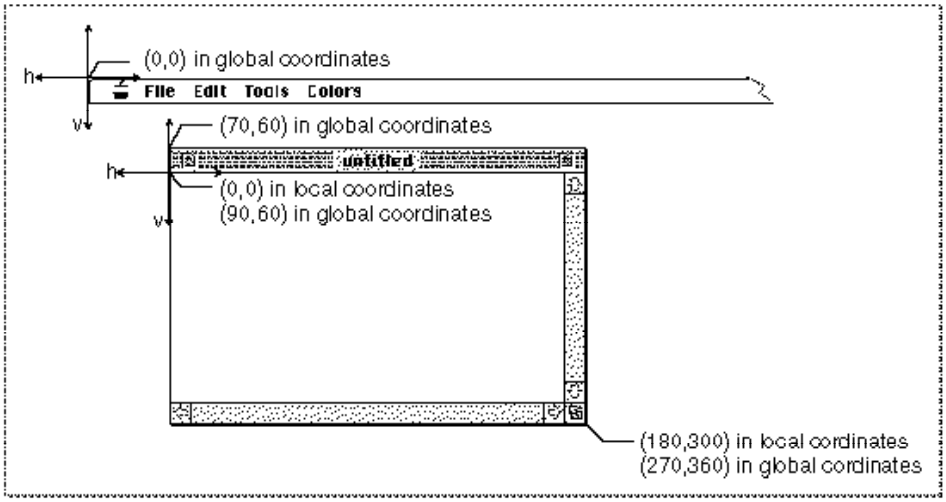
Note
When you place a window on the screen, you specify the location of its content region, in global coordinates. Remember to allow space for the window's title bar. On the main screen, remember to leave space for the menu bar. In the Roman script system, both the standard document title bar and the menu bar are 20 pixels high. You can determine the height of the menu bar with the Menu Manager `GetMBarHeight` function. You can calculate the height of the title bar by comparing the top of the window's structure region with the top of the window's content region. See Listing 4-12 on page 4-55 for a sample procedure that considers the menu bar and title bar when placing a window on the screen. ◆

Within the port rectangle, the drawing area is described in **local coordinates**—that is, in the coordinate system defined by the port rectangle. You draw into a window in local coordinates, without regard to the window's location on the screen (which is described in global coordinates). Figure 4-14 illustrates the local and global coordinate systems for a sample window 180 pixels high by 300 pixels wide, placed with its content region 70 pixels down and 60 pixels to the right of the upper-left corner of the screen.

When the Window Manager creates a window, it places the origin of the local coordinate system at the upper-left corner of the window's port rectangle. You can redefine the coordinates of the port rectangle's upper-left corner with the QuickDraw procedure `SetOrigin`.

The Event Manager describes mouse events in global coordinates, and you do most of your window manipulation in global coordinates. You generally display user data and manipulate your controls in local coordinates. When you need to convert between the two, you can call the QuickDraw functions `GlobalToLocal` and `LocalToGlobal`, described in *Inside Macintosh: Imaging*.

Figure 4-14 A window's local and global coordinate systems



Window Records

Each window has a number of descriptive characteristics such as a title, control list, and visibility status. The Window Manager stores this information in a **window record,** which is a data structure of type `WindowRecord`.

The window record includes

- the window's graphics port data structure
- the window's class, which specifies whether it was created directly through the Window Manager or indirectly through the Dialog Manager
- the window title
- a series of flags that specify whether the window is visible, whether it's highlighted, whether it has a zoom box, and whether it has a close box
- pointers to the structure, content, and update regions
- a handle to the window's definition function
- a handle to the window's control list
- an optional handle to a picture of the window's contents
- a reference constant field that your application can use as needed

The window record is described in detail in "The Color Window Record" beginning on page 4-65.

The first field in the window record is the window's graphics port. The `WindowPtr` data type is therefore defined as a pointer to a graphics port.

```
TYPE  WindowPtr   =  GrafPtr;
```

You draw into a window by drawing into its graphics port, passing a window pointer to the QuickDraw drawing routines. You also pass window pointers to most Window Manager routines.

You don't usually need to access or directly modify fields in a window record. When you do, however, you can refer to them through the `WindowPeek` data type, which is a pointer to a window record.

```
TYPE  WindowPeek  =  ^WindowRecord;
```

The close box, drag region, zoom box, and size box are not included in the window record because they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are.

Your application seldom accesses a window record directly; the Window Manager automatically updates the window record when you make any changes to the window, such as changing its title. The Window Manager also supplies routines for changing and reading some parts of the window record.

Color Windows

Since the introduction of Color QuickDraw, the Window Manager has supported color windows. Color windows are displayed in color graphics ports, as described in *Inside Macintosh: Imaging*. The color window record is exactly like the window record described in "Window Records" on page 4-19, except that it contains a color graphics port instead of a monochrome graphics port.

Whether or not your application uses color explicitly, and whether or not a color monitor is currently installed, your application should work with color windows whenever Color QuickDraw is available. Once you have created a window, you can use the window record and window pointer for a color window interchangeably with the window record and window pointer for a monochrome window.

On a monitor that is set to display 4-bit color or greater, the Window Manager automatically displays the window title and parts of the frame and controls in color (or gray scale, depending on the capabilities of the monitor). The user can adjust these colors through the Color control panel. Except in unusual circumstances, your application should not try to change the colors of the window frame. On a monitor that's set to display 1-bit color, the Window Manager draws the window title, frame, and controls in black and white.

Various elements of a window's colors are controlled by the **window color table,** which contains a series of part codes for different window elements and the RGB values associated with each part.

Because the user can select window display colors for the entire desktop, and because the Window Manager performs some complex display calculations automatically if you don't override it, your application typically uses the default system window color table.

If your application explicitly controls the colors used in a window, however, you can define your own window color tables.

You define a window color table for a window by providing a window color table resource (that is, a resource of type `'wctb'`) with the same resource ID as the window's `'WIND'` resource. The Window Manager creates a window color table when it creates the window record. The Window Manager maintains its own linked list, using **auxiliary window records,** which associates your application's windows with their corresponding window color tables. The window color table and the auxiliary window record are described in "The Window Color Table Record" beginning on page 4-71 and "The Auxiliary Window Record" beginning on page 4-73.

Except in unusual circumstances, your application doesn't need to manipulate window color tables or the auxiliary window record.

For compatibility with other applications in the shared environment, your application should not manipulate system color tables directly but should use the Palette Manager, as described in *Inside Macintosh: Imaging.* If your application provides its own window and control definition functions, they should apply the user's desktop color choices just as the default definition functions do.

Events in Windows

Events are messages that describe user actions and report on the processing status of your application. The Window Manager generates two kinds of events: activate events and update events. Activate events tell your application that a specified window is becoming active or inactive. Update events tell your application that it must redraw part or all of a window's content region. The section "Handling Events in Windows" beginning on page 4-41 describes when these events occur and how your application responds.

One of the basic functions of the Window Manager is to report where the cursor is when your application receives a mouse-down event. The Window Manager function `FindWindow` tells your application whether the cursor is in a window and, if it's in a window, which window it's in and where in that window (that is, the title bar, the drag region, and so on). You can use the `FindWindow` function as a first filter for mouse-down events, separating events that merely affect the window display from events that manipulate user data.

The Window Manager also provides a set of routines that help you implement the standard window-manipulation conventions:

User action	Application response
Dragging the title bar	Moves the window
Dragging the size box	Resizes the window
Clicking the zoom box	Toggles the window between two sizes and locations, known as the *user state* and the *standard state*
Clicking the close box	Closes the window

The next section, "Using the Window Manager," describes how you can use the Window Manager to move, resize, zoom, and close windows.

You can call the Control Manager to handle events in window controls, as described in the chapter "Control Manager" in this book. If you use the Dialog Manager for your alert boxes and modal dialog boxes, the Dialog Manager handles keyboard activity and mouse events in these windows. You can also use the Dialog Manager to handle keyboard activity and mouse events in the content region of movable modal dialog boxes and modeless dialog boxes. Your application, however, must handle mouse events in the title bar and close box of a movable modal or modeless dialog box.

When your application is active, a mouse-down event in a window belonging to any other application, including the Finder, switches your application to the background (unless there's an alert box or a modal dialog box pending, in which case the Dialog Manager merely sounds the system alert).

Using the Window Manager

Virtually every Macintosh application uses the Window Manager, both to simplify the display and management of windows and to retrieve basic information about user activities.

Your application works in conjunction with the Window Manager to present the standard user interface for windows. When the user clicks in an inactive window belonging to your application, for example, you can call the procedure `SelectWindow`, which highlights the newly active window, removes the highlighting from the previously active window, and generates the activate events that trigger the activation and deactivation of the two affected windows.

Your application can also use Window Manager routines to handle direct window manipulation. For example, if the user presses the mouse button when the cursor is in the title bar of a window, you can call the `DragWindow` procedure to track the mouse and drag an outline of the window on the screen until the user releases the mouse button.

You typically create windows from window resources, which are resources of type `'WIND'`. The Window Manager supports the nine types of windows described in "Types of Windows" beginning on page 4-8. (You can also write your own window definition functions to support your own window types. Window definition functions are stored as resources of type `'WDEF'`.) Alert box windows and dialog box windows use alert (`'ALRT'`), dialog (`'DLOG'`), and item list (`'DITL'`) resources; the chapter "Dialog Manager" describes how to create these resources. Most windows contain controls, which are defined through control (`'CNTL'`) resources; the chapter "Control Manager" describes how to create control resources.

Your application typically uses the Window Manager in conjunction with both the Control Manager and the Dialog Manager. You use the Control Manager to define, draw, and manipulate controls in your windows. If your window includes scroll bars, for example, you can use the `TrackControl` function to track the mouse while the user drags the scroll box. You can use the Dialog Manager to create, display, and track events in alert boxes and dialog boxes.

System 7 provides help balloons for the window frame—that is, the title bar, zoom box, and close box—of a window created with one of the standard window definition functions. You should provide help balloons for your window content region—that is, the size box, controls, and data area—and for the window frames of any window types you define. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for a description of how to use help balloons.

Before using the Window Manager, you must call the procedure `InitGraf` to initialize QuickDraw, the procedure `InitFonts` to initialize the Font Manager, and finally the procedure `InitWindows` to initialize the Window Manager.

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.

When your application receives an event, it responds according to what kind of window is currently active and where the event occurred. When it receives a mouse-down event in the content region of an active document window, your application follows its own conventions: inserting text, making a selection, or adding graphics, for example. When it receives a mouse-down event in the menu bar, your application enables and disables menu items as appropriate—which again depends on what kind of window is active and what is selected in that window. If the user has the insertion point in an editable text field in a modal dialog box, for example, the only menu item available might be Paste in the Edit menu—and then only if there is something in the scrap to be pasted.

You can use various strategies for keeping track of different kinds of windows. The `refCon` field in the window record is set aside specifically for use by applications. You can use the `refCon` field to store different kinds of data, such as a number that represents a window type or a handle to a record that describes the window.

The sample code in this chapter—excerpts from the SurfWriter application used throughout this book—uses a hybrid strategy:

- For document windows, the `refCon` field holds a handle to a document record.
- For modeless or movable modal dialog boxes, the `refCon` field holds a number that represents a type of dialog box.

You may well find other approaches more practical.

The SurfWriter application stores document information about the user's data, the window display, and the file, if any, associated with the data in a document record. The document record takes this form:

```
TYPE MyDocRec =
   RECORD
      editRec:       TEHandle;      {handle to text being edited}
      vScrollBar:    ControlHandle; {control handle to the }
                                    { vertical scroll bars}
      hScrollBar:    ControlHandle; {control handle to the }
                                    { horizontal scroll bars}
      fileRefNum:    Integer;       {reference number for file}
      fileFSSpec:    FSSpec;        {FSSpec record for file}
      windowDirty:   Boolean;       {whether data has changed }
                                    { since last save}
   END;
   MyDocRecPtr       = ^MyDocRec;
   MyDocRecHnd       = ^MyDocRecPtr;
```

The SurfWriter application creates a document record every time it creates a document window, and it stores a handle to the document record in the `refCon` field of the window record. (See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for a more complete illustration of how to use document records.)

When SurfWriter creates a modeless dialog box or a movable modal dialog box, it stores a constant that represents that dialog box (that is, it specifies the constant in the dialog resource, and the Window Manager sets the `refCon` field to that value when it creates the window record). For example, a `refCon` value of 20 might specify a modeless dialog box that accepts input for the Find command, and a value of 21 might specify a modeless dialog box that accepts input for the spelling checker.

When SurfWriter receives notification of an event in one of its windows, it first determines the function of the window and then dispatches the event as appropriate. Listing 4-1 illustrates an application-defined routine `MyGetWindowType` that determines the window's type.

Note

The `MyGetWindowType` function determines the type of a window from among a set of application-defined window types, which reflect the different kinds of windows the application creates. These window types are different from the standard window types defined by the definition functions, which determine how windows look and behave. To find out which one of the standard window types a window is, call the Window Manager function `GetWVariant`. ◆

The sample code later in this chapter calls the `MyGetWindowType` function as part of its event-handling procedure, described in the section "Handling Events in Windows" beginning on page 4-41.

Listing 4-1 Determining the window type

```
FUNCTION MyGetWindowType (thisWindow: WindowPtr): Integer;
VAR
   myWindowType:  Integer;
BEGIN
  IF thisWindow <> NIL THEN
  BEGIN
    myWindowType := WindowPeek(thisWindow)^.windowKind;
    IF myWindowType < 0 THEN                {window belongs to }
      MyGetWindowType := kDAWindow          { a desk accessory}
    ELSE
      IF myWindowType = userKind THEN       {document window}
        MyGetWindowType := kMyDocWindow
      ELSE                                  {dialog window}
         MyGetWindowType := GetWRefCon(window);    {get dialog ID}
  END
  ELSE
   MyGetWindowType := kNil;
END;
```

Notice that `MyGetWindowType` checks whether the window belongs to a desk accessory. This step ensures compatibility with older versions of system software. When your application is running in System 7, it should receive events only for its own windows and for windows belonging to desk accessories that were launched in its partition. See *Inside Macintosh: Memory* for information about partitions and *Inside Macintosh: Processes* for information about launching applications and desk accessories.

Creating a Window

You typically specify the characteristics of your windows—such as their initial size, location, title, and type—in window (`'WIND'`) resources. Once you have defined your window resources, you can call the function `GetNewCWindow` (or `GetNewWindow`) to create windows.

Defining a Window Resource

You typically define a window resource for each type of window that your application creates. If, for example, your application creates both document windows and special-purpose windows, you would probably define two window resources. Defining your windows in window resources lets you localize your window titles for different languages by changing only the window resources. (You specify the characteristics of alert boxes and dialog boxes with the alert and dialog resources, described in the chapter “Dialog Manager” in this book.)

Listing 4-2 shows a window resource, in Rez input format, that an application might use to create a document window. The resource specifies the attributes for windows created from the resource of type `'WIND'` with resource ID 128. The system software loads the resource into memory immediately after opening the resource file, and the Memory Manager can purge the memory occupied by the resource.

Listing 4-2 Rez input for a window (`'WIND'`) resource for a document window

```
#define rDocWindow      128

resource 'WIND' (rDocWindow, preload, purgeable) {
      {64, 60, 314, 460},  /*initial window size and location*/
      zoomDocProc,         /*window definition ID: */
                           /* incorporates definition function */
                           /* and variation code*/
      invisible,           /*window is initially invisible*/
      goAway,              /*window has close box*/
      0x0,                 /*reference constant*/
      "untitled",          /*window title*/
      staggerParentWindowScreen
                           /*optional positioning specification*/
};
```

The four numbers in the first element of this resource specify the upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional positioning code described later in this section. When specifying a window's position on the desktop, remember to leave room for the window's frame and, on the main screen, for the menu bar.

The second element contains the window's definition ID, which specifies both the window definition function that will handle the window and an optional variation code that defines a window type. If you are using one of the standard window types (described in "Types of Windows" beginning on page 4-8), you need to specify only one of the window-type constants listed in "The Window Resource" beginning on page 4-124.

The third element in the window resource specifies whether the window is initially visible or invisible. This element determines only whether the Window Manager displays the window when it first creates it, not whether the window can be seen on the screen. (A window entirely covered by other windows, for example, might be "visible," even though the user cannot see it.) You typically create new windows in an invisible state, build the content area of the window, and then display the completed window by calling `ShowWindow` to make it visible.

The fourth element in the window resource specifies whether the window has a close box. Only some of the standard window types (`zoomDocProc`, `noGrowDocProc`, `documentProc`, `zoomNoGrow`, and `rDocProc`) support close boxes. The close-box element has no effect if the second field of the resource specifies a window type that does not support a close box. The Window Manager draws the close box when it draws the window frame.

The fifth element in the window resource is a reference constant, in which your application can store whatever data it needs. When it builds a new window record, the Window Manager stores in the `refCon` field whatever value you specify here. You can also put a placeholder here (such as `0x0`, in this example) and then set the `refCon` field yourself by calling the `SetWRefCon` procedure.

The sixth element in the window resource is a string that specifies the window title.

The optional seventh element in the window resource specifies a positioning rule that overrides the window position specified by the rectangle in the first element. In the window resource for a document window, you typically specify the positioning constant `staggerParentWindowScreen`. For a complete list of the positioning constants and their effects, see "The Window Resource" beginning on page 4-124.

The positioning constants are convenient when the user is creating a new document or when you're handling your own dialog boxes and alert boxes. When you're creating a new window to display a previously saved document, however, the new window should appear, if possible, in the same rectangle as the previous window (that is, the window used during the last save). For the rules of window placement, see "Positioning a Document Window on the Desktop" beginning on page 4-30.

Use the function `GetNewCWindow` or `GetNewWindow` to create a window from a `'WIND'` resource.

Creating a Window From a Resource

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.

You create document windows from a window resource using the function `GetNewCWindow` or `GetNewWindow`. (Whenever Color QuickDraw is available, use `GetNewCWindow` to create color windows, whether or not a color monitor is currently installed. A color window record is the same size as a window record, and `GetNewCWindow` returns a pointer of type `WindowPtr`, so most code can handle color windows and monochrome windows identically.)

You can allow `GetNewCWindow` to allocate the memory for your window record. You can maintain more control over memory use, however, by allocating the memory yourself from a block allocated for such purposes during your own initialization routine, and then passing the pointer to `GetNewCWindow`.

You typically create the scroll bars from control (`'CNTL'`) resources at the time that you create a document window and then display them when you make the window visible.

Listing 4-3 illustrates an application-defined procedure, DoNewCmd, which SurfWriter calls when the user chooses New from the File menu. Windows are typically invisible when created and displayed only after all elements are in place.

Listing 4-3 Creating a new window

```
PROCEDURE DoNewCmd (newDocument: Boolean; VAR window: WindowPtr);
VAR
   myData:        MyDocRecHnd;   {the document's data record}
   windStorage:   Ptr;           {memory for window record}
   destRect,                     {rectangles for creating }
   viewRect:      Rect;          { TextEdit edit record}
   good:          Boolean;       {success flag}
BEGIN
   window := NIL;                   {no window created yet}
   good := FALSE;                   {no success yet}
   {allocate memory for window record from previously allocated block}
   windStorage := MyPtrAllocationProc;
   IF windStorage <> NIL THEN       {memory allocation succeeded}
   BEGIN                            {create window}
      IF gColorQDAvailable THEN
         window := GetNewCWindow(rDocWindow, windStorage, WindowPtr(-1))
      ELSE
         window := GetNewWindow(rDocWindow, windStorage, WindowPtr(-1));
   END;
                                    {create document record}
   myData := MyDocRecHnd(NewHandle(SIZEOF(MyDocRec)));
   IF (window <> NIL) AND (myData <> NIL) THEN  {window record and document }
   BEGIN                                        { record both allocated}
      SetPort(window);              {set current port}
      HLock(Handle(myData));        {lock handle to doc record}
      SetWRefCon(window, LongInt(myData));   {link document record to window}
      WITH window^, myData^^ DO     {fill in document record}
      BEGIN
         MyGetTERect(window, viewRect); {set up a viewRect for TextEdit}
         destRect := viewRect;
         destRect.right := destRect.left + kMaxDocWidth;
         editRec := TENew(destRect, viewRect);
         IF editRec <> NIL THEN           {it's a good edit record}
         BEGIN
            good := TRUE;                    {set success flag}
            MyAdjustViewRect(editRec);       {set up edit record}
            TEAutoView(TRUE, editRec);
         END
```

```
         ELSE
            good := FALSE;             {clear success flag}
         IF good THEN
         BEGIN                         {create scroll bars}
            vScrollBar := GetNewControl(rVScroll, window);
            hScrollBar := GetNewControl(rHScroll, window);
            good := (vScrollBar <> NIL) AND (hScrollBar <> NIL);
         END;
         IF good THEN                  {it's a good document}
         BEGIN
            MyAdjustScrollBars(window, FALSE);  {adjust scroll bars}
            fileRefNum := 0;           {no file yet}
            windowDirty := FALSE;      {no changes yet}
            IF newDocument THEN        {if it's a new (empty) document, }
               ShowWindow(window);     { make it visible}
         END;
      END;     {end of WITH statement}
      HUnlock(Handle(myData));         {unlock document record}
   END;     {end of IF (window <> NIL) AND (myData <> NIL)}
   IF NOT good THEN
   BEGIN
      IF windStorage <> NIL THEN  {memory for window record was allocated}
         DisposePtr(windStorage); {dispose of it}
      IF myData <> NIL THEN       {memory for document record was allocated}
      BEGIN
         IF myData^^.editRec <> NIL THEN  {edit record was allocated}
            TEDispose(myData^^.editRec);  {dispose of it}
         DisposeHandle(Handle(myData));   {dispose of document record}
      END;
      IF window <> NIL THEN      {window pointer exists, but it's invalid}
         CloseWindow(window);    {clean up window pointer}
      window := NIL;             {set window to NIL to indicate failure}
   END;
END; {DoNewCmd}
```

The `DoNewCmd` procedure first sets the window pointer and success flags to show that a valid window doesn't yet exist. Then it calls the application-defined function `MyPtrAllocationProc`, which allocates memory for a window record from a block set aside during program initialization for that purpose. If `MyPtrAllocationProc` successfully allocates memory and returns a valid pointer, `DoNewCmd` creates a window, specifying the `'WIND'` resource with resource ID 128, as specified by the constant `rDocWindow`. Using this window resource (defined in Listing 4-2 on page 4-26), the Window Manager creates an invisible window of type `zoomDocProc`. Because the `behind` parameter to `GetNewCWindow` or `GetNewWindow` has the value `WindowPtr(–1)`, the Window Manager places the new window in front of all others on the desktop.

The DoNewCmd procedure then creates a document record. It locks the document record in memory while manipulating it, sets the refCon field in the window record so that it points to the document record, and fills in the document record. While filling in the document record, DoNewCmd sets up a TextEdit record to hold the user's data. If that succeeds, DoNewCmd sets up horizontal and vertical scroll bars. If that succeeds, DoNewCmd adjusts the scroll bars (see the chapter "Control Manager" in this book for the application-defined procedure MyAdjustScrollbars) and fills in the remaining parts of the document record. If the window is being created to display a new document, that is, if no user data needs to be read from a disk, DoNewCmd calls the ShowWindow procedure to make the window visible immediately.

If your window resource specifies that a new window is visible, GetNewCWindow displays the window immediately. If you're creating a document window, however, you're more likely to create the window in an invisible state and then make it visible when you're ready to display it.

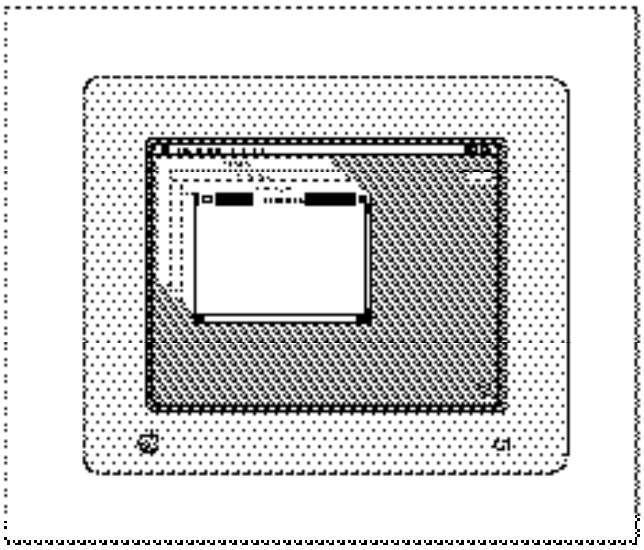
- If you're creating a window because the user is creating a new document, you can display the window immediately by calling the procedure ShowWindow to make the window frame visible. This change in visibility adds to the update region and triggers an update event. Your application then invokes its own procedure for drawing the content region in response to the update event.
- If you're creating a new window to display a saved document, you must retrieve the user's data before displaying it. (See *Inside Macintosh: Files* for information about reading saved files.) If possible, the size and location of the window that displays the document should be the same as when the document was last saved. (See the next section, "Positioning a Document Window on the Desktop," for a discussion of window placement.) Once you have positioned the window and set up its content region, you can make the window visible by calling ShowWindow, which triggers an update event. Your application then invokes its own procedure for drawing the content region.

Positioning a Document 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). This section describes the placement of document windows. The chapter "Dialog Manager" in this book describes the placement of alert boxes and dialog boxes. See *Macintosh Human Interface Guidelines* for a complete description of window placement.

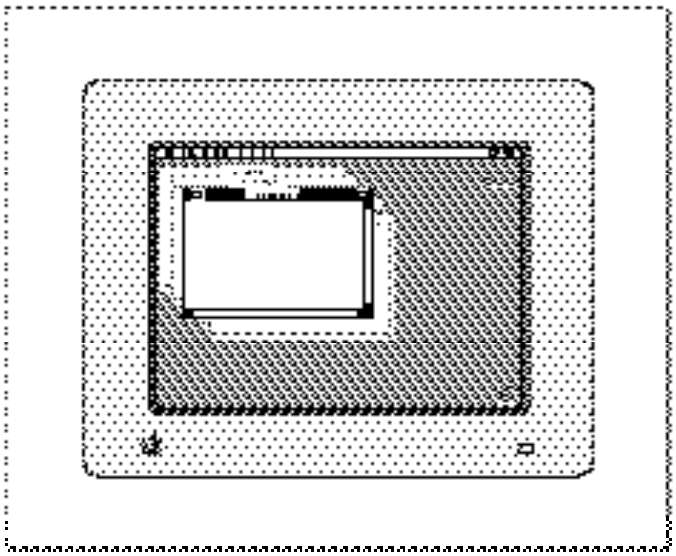
On Macintosh computers with a single screen of known size, positioning windows is fairly straightforward. You position the first new document window on the upper-left corner of the desktop. Open each additional new document window with its upper-left corner slightly below and to the right of the upper-left corner of its predecessor. Figure 4-15 illustrates how to position multiple documents on a single screen.

Figure 4-15 Document window positions on a single screen



If the user closes one or more document windows, display subsequent windows in the "empty" positions before adding more positions below and to the right. Figure 4-16 illustrates how you fill in an empty position when the user opens a new document after closing one created earlier.

Figure 4-16 "Filling in" an empty document window position



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 in the window resource that let you position new windows automatically. You typically use the constant `staggerParentWindowScreen` for positioning document windows. The `staggerParentWindowScreen` constant specifies the basic guidelines for document window placement: When creating windows from a template that includes `staggerParentWindowScreen`, 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 20 pixels to the right and 20 pixels below the upper-left corner of the last window in which the user was working. Figure 4-17 illustrates how the Window Manager positions a new document window when the `staggerParentWindowScreen` specification is in effect and the user has been working in a window off the main screen.

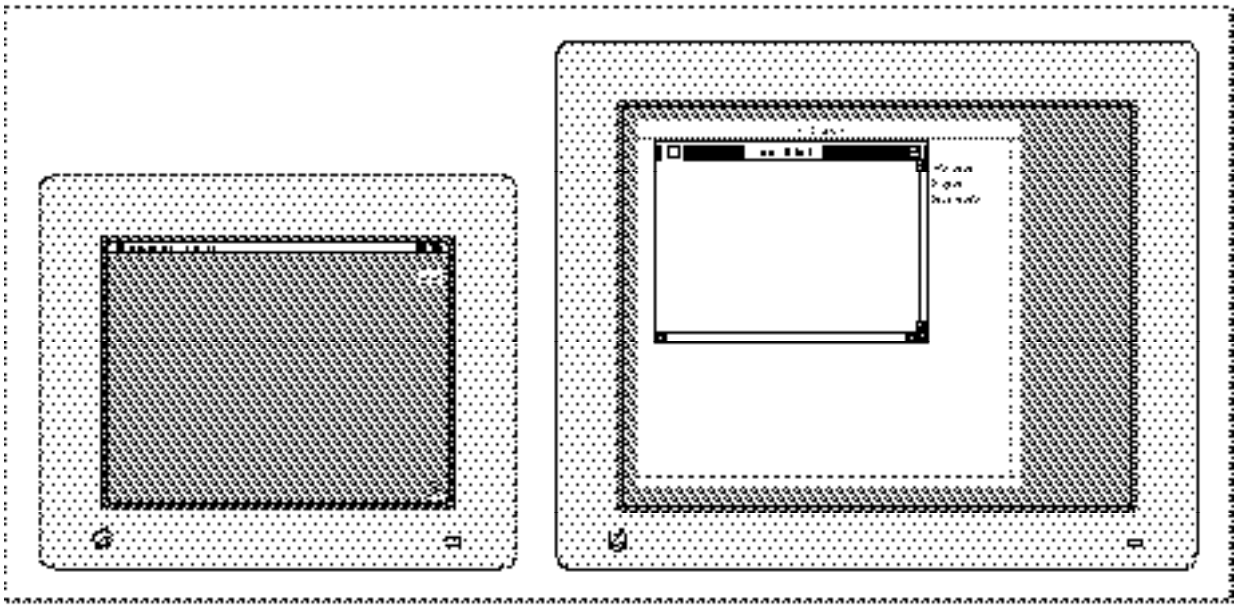
If the user moves or closes a window that occupies one of the interim positions, and the window template specifies `staggerParentWindowScreen`, the Window Manager uses the “empty” slot for the next new window created before moving further down and to the right.

For a complete list of the positioning constants and their effects, see “The Window Resource” beginning on page 4-124.

You can usually use the `staggerParentWindowScreen` positioning constant when creating a window that is to display a new document. You must perform your own window-placement calculations, however, when opening saved documents and when zooming windows.

When the user saves a document, the document window can be in one of two states: the user state or the standard state.

Figure 4-17 Document window positions on multiple screens



The **user state** is the last size and location the user established for the window.

The **standard state** is what your application determines is the most convenient size for the window, considering the function of the document and the screen space available. For a more complete description of the standard state, see "Zooming a Window" beginning on page 4-53. Your application typically calculates the standard state each time the user zooms to that state.

The user and standard states are stored in the state data record, whose handle appears in the `dataHandle` field of the window record.

```
TYPE WStateData =
   RECORD
      userState:  Rect;    {size and location established by user}
      stdState:   Rect;    {size and location established by }
                           { application}
   END;
```

When the user saves a document, you must save the user state rectangle and the state of the window (that is, whether the window is in the user state or the standard state). Then, when the user opens the document again later, you can replicate the window's status. You typically store the state data as a resource in the resource fork of the document file.

Listing 4-4 illustrates an application-defined data structure for storing the window's user rectangle and state.

Listing 4-4 Application-defined data structure for storing a window's state data

```
TYPE MyWindowState =
   RECORD
      userStateRect: Rect;    {user state rectangle}
      zoomState:     Boolean; {window state: TRUE = standard; }
                              {              FALSE = user}
   END;

MyWindowStatePtr = ^MyWindowState;
MyWindowStateHnd = ^MyWindowStatePtr;
```

This structure translates into an application-defined resource that is stored in the resource fork of the document when the user saves the document.

Listing 4-5 shows an application-defined routine for saving a document's state data. The SurfWriter application calls the procedure `MySaveWindowPosition` when the user saves a document.

Listing 4-5 Saving a document window's position

```
PROCEDURE MySaveWindowPosition (myWindow: WindowPtr;
                                myResFileRefNum: Integer);
VAR
   lastWindowState:  MyWindowState;
   myStateHandle:    MyWindowStateHnd;
   curResRefNum:     Integer;
BEGIN
   {Set user state provisionally and determine whether window is zoomed.}
   lastWindowState.userStateRect := WindowPeek(myWindow)^.contRgn^^.rgnBBox;
   lastWindowState.zoomState := EqualRect(lastWindowState.userStateRect,
                                          MyGetWindowStdState(myWindow));
   {if window is in standard state, then set the window's user state from }
   { the userState field in the state data record}
   IF lastWindowState.zoomState THEN      {window was in standard state}
      lastWindowState.userStateRect := MyGetWindowUserState(myWindow);
   curResRefNum := CurResFile;   {save the refNum of current resource file}
   UseResFile(myResFileRefNum);  {set the current resource file}
   myStateHandle := MyWindowStateHnd(Get1Resource(rWinState,
                                                  kLastWinStateID));
```

```
   IF myStateHandle <> NIL THEN        {a state data resource already exists}
   BEGIN                               {update it}
      myStateHandle^^ := lastWindowState;
      ChangedResource(Handle(myStateHandle));
   END
   ELSE                                {no state data has yet been saved}
   BEGIN                               {add state data resource}
      myStateHandle := MyWindowStateHnd(NewHandle(SizeOf(MyWindowState)));
      IF myStateHandle <> NIL THEN
      BEGIN
         myStateHandle^^ := lastWindowState;
         AddResource(Handle(myStateHandle), rWinState, kLastWinStateID,
                     'last window state');
      END;
   END;
   IF myStateHandle <> NIL THEN
   BEGIN
      UpdateResFile(myResFileRefNum);
      ReleaseResource(Handle(myStateHandle));
   END;
   UseResFile(curResRefNum);
END;
```

The `MySaveWindowPosition` procedure first determines whether the window is in the user state or the standard state by setting its own user state field from the bounding rectangle of the window's content region and comparing that rectangle with the user state stored in the state data record. (If the two match, the window is in the user state; if not, the standard state.) If the window is in the standard state, the procedure replaces its own user state data with the rectangle stored in the `userState` field of the state data record. The rest of the procedure saves the application-defined state data record in the resource fork of the document.

When creating a new window to display a saved document, SurfWriter restores the saved user state data and recalculates the standard state. Before using the saved rectangle, however, SurfWriter verifies that the location is reachable on the desktop. (If the user saves a document on a computer equipped with multiple monitors and then opens it later on a system with only one monitor, for example, the saved window location could be entirely or partially off the screen.)

Listing 4-6 on the next page shows `MySetWindowPosition`, the application-defined routine that SurfWriter calls when the user opens a saved document. The `MySetWindowPosition` procedure retrieves the document's saved state data and then calls another application- defined routine, `MyVerifyPosition`, to verify that the saved location is practical.

Listing 4-6 Positioning the window when the user opens a saved document

```
PROCEDURE MySetWindowPosition (myWindow: WindowPtr);
VAR
   myData:              MyDocRecHnd;
   lastUserStateRect:   Rect;
   stdStateRect:        Rect;
   curStateRect:        Rect;
   myRefNum:            Integer;
   myStateHandle:       MyWindowStateHnd;
   resourceGood:        Boolean;
   savePort:            GrafPtr;
   myErr:               OSErr;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(myWindow));    {get document record}
   HLock(Handle(myData));        {lock the record while manipulating it}
   {open the resource fork and get its file reference number}
   myRefNum := FSpOpenResFile(myData^^.fileFSSpec, fsRdWrPerm);
   myErr := ResError;
   IF myErr <> noErr THEN
      Exit(MySetWindowPosition);
   {get handle to rectangle that describes document's last window position}
   myStateHandle := MyWindowStateHnd(Get1Resource(rWinState,
                                                  kLastWinStateID));
   IF myStateHandle <> NIL THEN                {handle to data succeeded}
   BEGIN    {retrieve the saved user state}
      lastUserStateRect := myStateHandle^^.userStateRect;
      resourceGood := TRUE;
   END
   ELSE
   BEGIN
      lastUserStateRect.top := 0;   {force MyVerifyPosition to calculate }
      resourceGood := FALSE;        { the default position}
   END;
   {verify that user state is practical and calculate new standard state}
   MyVerifyPosition(myWindow, lastUserStateRect, stdStateRect);
   IF resourceGood THEN                   {document had state resource}
      IF myStateHandle^^.zoomState THEN   {if window was in standard state }
         curStateRect := stdStateRect     { when saved, display it in }
                                          { newly calculated standard state}
      ELSE                {otherwise, current state is the user state}
         curStateRect := lastUserStateRect
   ELSE                                   {document had no state resource}
      curStateRect := lastUserStateRect;  {use default user state}
```

```
   {move window}
   MoveWindow(myWindow, curStateRect.left, curStateRect.top, FALSE);
   {Convert to local coordinates and resize window.}
   GetPort(savePort);
   SetPort(myWindow);
   GlobalToLocal(curStateRect.topLeft);
   GlobalToLocal(curStateRect.botRight);
   SizeWindow(myWindow, curStateRect.right, curStateRect.bottom, TRUE);
   IF resourceGood THEN       {reset user state and standard }
   BEGIN                   { state--SizeWindow may have changed them}
      MySetWindowUserState(myWindow, lastUserStateRect);
      MySetWindowStdState(myWindow, stdStateRect);
   END;
   ReleaseResource(Handle(myStateHandle));         {clean up}
   CloseResFile(myRefNum);
   HUnLock(Handle(myData));
END;
```

The `MyVerifyPosition` routine, not shown here, compares the saved location against available screen space. (See Listing 4-12 on page 4-55 for a strategy for comparing the saved rectangle with the available screen space.) `MyVerifyPosition` alters the user state rectangle, if necessary (using the same size, if possible, but placing it on available screen space) and calculates a new standard state for displaying the window on the screen containing the user state.

After determining valid user and standard state rectangles, the procedure `MySetWindowPosition` sets a temporary positioning rectangle to the appropriate size and location, based on the state of the document's window when the document was saved. The `MySetWindowPosition` procedure then calls the Window Manager procedures `MoveWindow` and `SizeWindow` to establish the window's location and size before cleaning up.

The SurfWriter application calls `MySetWindowPosition` from its routine for opening saved documents, after reading the document's data from its data fork. Listing 4-7 shows the application-defined `DoOpenFile` function that SurfWriter calls when the user opens a saved document.

Listing 4-7 Opening a saved document

```
FUNCTION DoOpenFile (mySpec: FSSpec): OSErr;
VAR
   myWindow:      WindowPtr;
   myData:        MyDocRecHnd;
   myFileRefNum:  Integer;
   myErr:         OSErr;
```

```
BEGIN
   DoNewCmd(FALSE, myWindow);    {FALSE tells DoNewCmd not to }
                                 { show the window}
   IF myWindow = NIL THEN
   BEGIN
      DoOpenFile := kOpenFileError;
      Exit(DoOpenFile);
   END;
   SetWTitle(myWindow, mySpec.name);
   {open the file's data fork, passing the file spec-- }
   { FSpOpenDF returns a file reference number}
   myErr := FSpOpenDF(mySpec, fsRdWrPerm, myFileRefNum);
   IF (myErr <> noErr) AND (myErr <> opWrErr) THEN {open failed}
   BEGIN                                           {clean up}
      DisposeWindow(myWindow);
      DoOpenFile := myErr;
      Exit(DoOpenFile);
   END;
   {get a handle to the window's document record}
   myData := MyDocRecHnd(GetWRefCon(myWindow));
   myData^^.fileRefNum := myFileRefNum;   {save file ref num}
   myData^^.fileFSSpec := mySpec;         {save fsspec}
   myErr := DoReadFile(myWindow);         {read file's data}
   {retrieve saved state data and establish valid position}
   MySetWindowPosition(myWindow);
   {MyResizeWindow invalidates the whole portRgn, guaranteeing }
   { an update event--the window's contents are redrawn then}
   MyResizeWindow(myWindow);
   ShowWindow(myWindow);                  {show window}
   DoOpenFile := myErr;
END;
```

`DoOpenFile` first calls the application-defined procedure `DoNewCmd` to create a new window, suppressing the immediate display of the window. (Listing 4-3 on page page 4-28 illustrates the procedure `DoNewCmd`.) Then `DoOpenFile` sets the window title to the name of the document file and reads in the data. Then it calls `MySetWindowPosition` to determine where to place the new window. After establishing a valid position, `DoOpenFile` calls the application-defined routine `MyResizeWindow` (shown in Listing 4-14 on page 4-59) to set up the content region in the new dimensions, and then it finally makes the window visible.

Drawing the Window Contents

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 frame in the appropriate location. As the user makes changes to the desktop, moving and resizing different windows, the Window Manager alters the window frames as necessary. The window frame includes the window outline, the title bar, and the close and zoom boxes.

Your application is responsible for drawing the window's content region. It typically uses the Control Manager to draw the window controls, uses the Window Manager to draw the size box, and draws the user data itself. The sample code in this chapter uses the simple model of a content region that contains only controls, the size box, and a TextEdit record. (See *Inside Macintosh: Text* for a description of TextEdit.)

Listing 4-8 illustrates an application-defined procedure that draws the content region of a window.

Listing 4-8 Drawing a window

```
PROCEDURE MyDrawWindow (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
   BEGIN
      EraseRect(portRect);    {erase content area}
      UpdateControls(window, visRgn);  {draw window controls}
      DrawGrowIcon(window);   {draw size box}
      {update window contents as appropriate to your }
      { application (in this case use TextEdit)}
      TEUpdate(portRect, myData^^.editRec);
   END;
   HUnLock(Handle(myData));
END;
```

The `MyDrawWindow` procedure first sets the current port to the window's port and gets a handle to the window's document record. Using the data in the document record, the procedure first erases the content region, draws the controls, and draws the size box. Finally, it draws the user's data, in this case the contents of a TextEdit edit record.

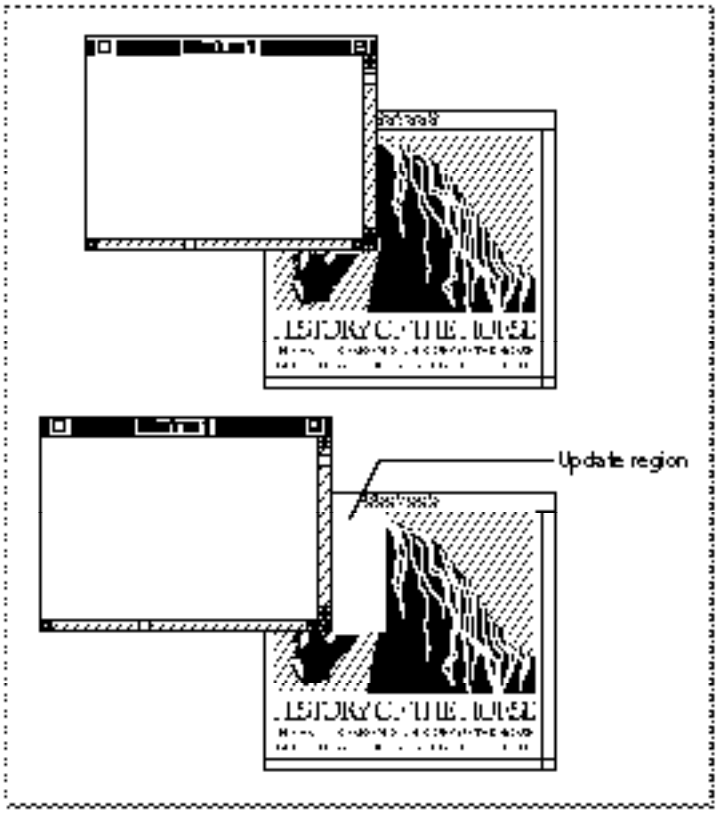
If your application creates a window that contains a static display, you can let the Window Manager take care of drawing and updating the content region by placing a handle to a picture in the `windowPic` field of the window record. See the description of the `SetWindowPic` procedure on page 4-110.

Updating the Content 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.

Figure 4-18 illustrates how the Window Manager adds part of a window's content region to its update region when the user exposes additional content area.

Figure 4-18 Moving one window and adding to another window's update region



The Event Manager periodically 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, as described in the section "Handling Update Events" beginning on page 4-48.

As the user makes changes to a document, your application must update both the document data and the document display in the content area of its window. You can use one of two strategies for updating the display:

- If your application doesn't require continuous scrolling or rapid response, you can add changed areas of the content region to the window's update region. The Event Manager then sends your application an update event, and your application invokes its standard update procedure.
- For continuous scrolling and a faster response time, you can draw directly into the content area of the window.

In either case, your application ultimately draws in the graphics port that represents the window. You draw controls through the Control Manager, and you draw text and graphics with the routines described in *Inside Macintosh: Text* and *Inside Macintosh: Imaging*.

Maintaining the Update Region

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

Your application usually manipulates the update region, for example, when the user resizes a window that contains a size box and scroll bars. If the user enlarges the window, the Window Manager adds the newly exposed area to the window's update region but does not add the area formerly occupied by the scroll bars. Before calling the `SizeWindow` procedure to resize the window, your application can call the `InvalRect` procedure twice to add the scroll bar and size box areas to the update region. The next time it receives an update event, your application erases the scroll bars and draws whatever parts of the document content might be visible at that location.

Similarly, you can remove an area from the update region when you know that it is in fact valid. Limiting the size of the update region decreases time spent redrawing. Listing 4-13 on page 4-58, for example, uses the `ValidRect` procedure to remove the unaffected text area from the update region of a window that is being resized.

Handling Events in Windows

Your application must be prepared to handle two kinds of window-related events:

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

In System 7 your application receives mouse-down events if it is the foreground process and the user clicks in the menu bar, a window belonging to your application, or a window belonging to a desk accessory that was launched in your application's partition. (If the user clicks in a window belonging to another application, the Event Manager sends your application a suspend event and performs a major switch to the other application—unless the frontmost window is an alert box or a modal dialog box, in which case the Dialog Manager merely sounds the system alert, and the Process Manager retains your application as the foreground process.) 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 routines, as described in the next section, "Handling Mouse Events in Windows."

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 "Handling Update Events" beginning on page 4-48.

Your application receives activate events when an inactive window becomes active or an active window becomes inactive. Activate events are an example of the close cooperation between your application and the Window Manager. When you receive a mouse-down event in one of your application's inactive windows, you can call the `SelectWindow` procedure, which removes the highlighting from the previously active window and adds highlighting to the newly active window. It also generates two activate events: one telling your application to deactivate the previously active window and one to activate the newly active window. Your application then activates and deactivates the content regions, as described in the section "Handling Activate Events" beginning on page 4-50.

When the user first clicks in an inactive window, most applications do not make a selection or otherwise change the window or document, beyond making the window active. When your application receives a resume event because the user clicked in one of its windows, you might not even want to receive the mouse-down event that caused your application to become the foreground process. You control whether or not you receive this event through the `'SIZE'` resource, described in the chapter "Event Manager" earlier in this book.

Handling Mouse Events in Windows

When your application is active, it receives notice of all keyboard activity and mouse-down events in the menu bar, in one of its windows, or in any windows belonging to desk accessories that were launched in its partition.

When it receives a mouse-down event, your application calls the `FindWindow` function to map the cursor location to a window region.

The function specifies the region by returning one of these constants:

```
CONST inDesk      = 0;  {none of the following}
      inMenuBar   = 1;  {in menu bar}
      inSysWindow = 2;  {in desk accessory window}
```

```
inContent   = 3;  {anywhere in content region except size }
                  { box if window is active, }
                  { anywhere including size box if window }
                  { is inactive}
inDrag      = 4;  {in drag (title bar) region}
inGrow      = 5;  {in size box (active window only)}
inGoAway    = 6;  {in close box}
inZoomIn    = 7;  {in zoom box (window in standard state)}
inZoomOut   = 8;  {in zoom box (window in user state)}
```

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 sets a variable parameter that points to the window.

In System 7, if `FindWindow` returns `inDesk`, the cursor is somewhere other than in the menu bar, one of your windows, or a window created by a desk accessory launched in your application's partition. The function may return `inDesk` if, for example, the cursor is in the window frame but not in the drag region, close box, or zoom box. `FindWindow` seldom returns the value `inDesk`, and you can generally ignore the rare instances of this function result.

If the user presses the mouse button with the cursor in the menu bar (`inMenuBar`), you call your own routines for displaying menus and allowing the user to choose menu items.

The `FindWindow` function returns the value `inSysWindow` only when the user presses the mouse button with the cursor in a window that belongs to a desk accessory launched in your application's partition. You can then call the `SystemClick` procedure, passing it the event record and window pointer. The `SystemClick` procedure, documented in the chapter "Event Manager" in this book, makes sure that the event is handled by the appropriate desk accessory.

The `FindWindow` function returns one of the other values when the user presses the mouse button while the cursor is in one of your application's windows. Your response depends on whether the cursor is in the active window and, if not, what kind of window is active.

When you receive a mouse-down event in the active window, you route the event to the appropriate routine for changing the window display or the document contents. When the user presses the mouse button while the cursor is in the zoom box, for example, you call the Window Manager function `TrackBox` to highlight the zoom box and track the mouse until the button is released.

When you receive a mouse-down event in an inactive window, your response depends on what kind of window is active:

- If the active window is a movable modal dialog box, you should sound the system alert and take no other action. (If the active window is a modal dialog box handled by the `ModalDialog` procedure, the Dialog Manager doesn't pass the event to your application but sounds the system alert itself.)

- If the active window is a document window or a modeless dialog box, you can call `SelectWindow`, passing it the window pointer. The `SelectWindow` procedure removes highlighting from the previously active window, brings the newly activated window to the front, highlights it, and generates the activate and update events necessary to tell all affected applications which windows must be redrawn.

Listing 4-9 illustrates an application-defined procedure that handles mouse-down events.

Listing 4-9 Handling mouse-down events

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:       Integer;
   thisWindow: WindowPtr;
BEGIN
   part := FindWindow(event.where, thisWindow); {find out where cursor is}
   CASE part OF
   inMenuBar:           {cursor is in menu bar}
      BEGIN
         {make sure menu items are properly enabled/disabled}
         MyAdjustMenus;
         {let user choose a menu command}
         DoMenuCommand(MenuSelect(event.where));
      END;
   inSysWindow:            {cursor is in a desk accessory window}
      SystemClick(event, thisWindow);
   inContent:              {cursor is in the content region of one }
                           { of your application's windows}
      IF thisWindow <> FrontWindow THEN   {cursor is not in front window}
      BEGIN
         IF MyIsMovableModal(FrontWindow) THEN     {front window is }
            SysBeep(30)                            { movable modal}
         ELSE                          {front window is not movable modal}
            SelectWindow(thisWindow);  {make thisWindow active}
      END
      ELSE                 {cursor is in content region of active window}
         DoContentClick(thisWindow, event);  {handle event in content region}
   inDrag:                 {cursor is in drag area}
      {if a movable modal is active, ignore click in an inactive title bar}
      IF (thisWindow <> FrontWindow) AND MyIsMovableModal(FrontWindow) THEN
         SysBeep(30)
      ELSE
         {let Window Manager drag window}
         DragWindow(thisWindow, event.where, GetGrayRgn^^.rgnBBox);
   inGrow:                 {cursor is in size box}
      DoGrowWindow(thisWindow, event);    {change window size}
```

```
   inGoAway:               {cursor is in close box}
      {call TrackGoAway to handle mouse until button is released}
      IF TrackGoAway(thisWindow, event.where) THEN
         DoCloseCmd;                      {handle close window}
   inZoomIn, inZoomOut:    {cursor is in zoom box}
      {call TrackBox to handle mouse until button is released}
      IF TrackBox(thisWindow, event.where, part) THEN
         DoZoomWindow(thisWindow, part);  {handle zoom window}
   END;  {end of CASE statement}
END;     {end of DoMouseDownEvent}
```

The `DoMouseDown` procedure first calls `FindWindow` to map the location of the cursor to a part of the screen or a region of a window.

If the cursor is in the menu bar, `DoMouseDown` calls other application-defined procedures for adjusting and displaying menus and accepting menu choices.

If the cursor is in a window created by a desk accessory, `DoMouseDown` calls the `SystemClick` procedure, which handles mouse-down events for desk accessories from within applications.

If the cursor is in the content area of a window, `DoMouseDown` first checks to see whether the cursor is in the currently active window by comparing the window pointer returned by `FindWindow` with the result returned by the function `FrontWindow`. If the cursor is in an inactive window, `DoMouseDown` checks to see if the active window is a movable modal dialog box. (If the front window is an alert box or a fixed-position modal dialog box, an application does not receive mouse-down events in other windows.) If the active window is a movable modal dialog box and the cursor is in another window, `DoMouseDown` simply sounds the system alert and waits for another event. If the active window is not a movable modal dialog box, `DoMouseDown` calls `SelectWindow` to activate the window in which the cursor is located. The `SelectWindow` procedure relayers the windows as necessary, adjusts the highlighting, and sends the application a pair of activate events to deactivate the previously active window and activate the newly active window. `DoMouseDown` merely activates the window in which the cursor is located; it does not make a selection in the newly activated window in response to the first click in that window.

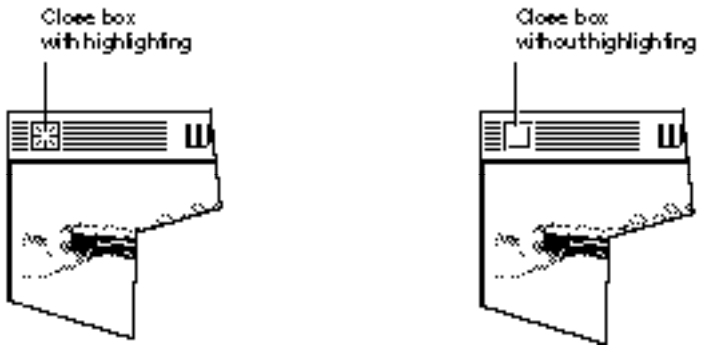
If the cursor is in the content area of the active window, the `DoMouseDown` procedure calls another application-defined procedure (`DoContentClick`) that handles mouse events in the content area.

If the cursor is in the drag region of a window, `DoMouseDown` first checks whether the drag region is in an inactive window while a movable modal dialog box is active. In that case, `DoMouseDown` merely sounds the system alert and waits for another event. In any other case, `DoMouseDown` calls the Window Manager procedure `DragWindow`, which displays an outline of the window, moves the outline as long as the user continues to drag the window, and calls `MoveWindow` to draw the window in its new location when the user releases the mouse button. After the window is drawn in its new location, it is the active window, whether or not it was active before.

If the cursor is in the size box, DoMouseDown calls another application-defined routine (DoGrowWindow, shown in Listing 4-13 on page 4-58) that resizes the window.

If the mouse press occurs in the close box, DoMouseDown calls the TrackGoAway function, which highlights the close box and tracks all mouse activity until the user releases the mouse button. As long as the user holds down the mouse button and leaves the cursor in the close box, TrackGoAway leaves the close box highlighted, as illustrated in Figure 4-19. If the user moves the cursor out of the close box, TrackGoAway removes the highlighting.

Figure 4-19 The close box with and without highlighting

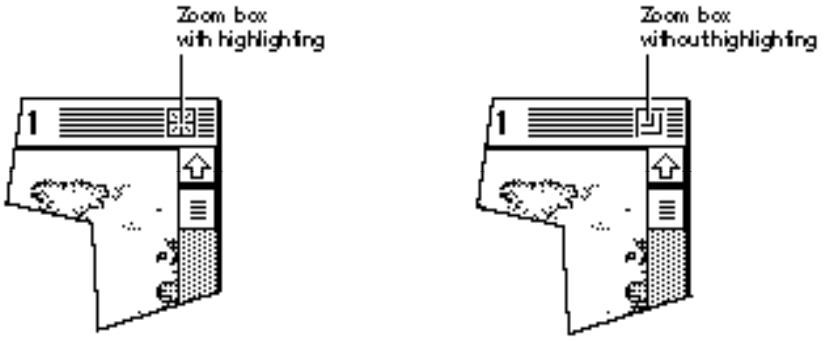


When the user releases the mouse button, TrackGoAway returns TRUE if the cursor is still in the close box and FALSE if it is not. If TrackGoAway returns TRUE, DoMouseDown calls the application-defined procedure DoCloseCmd to close the window. Listing 4-16 on page 4-60 shows the DoCloseCmd procedure.

If the mouse press occurs in the zoom box, the DoMouseDown procedure first calls TrackBox, which highlights the zoom box and tracks all mouse activity until the user releases the mouse button. As long as the user holds down the mouse button and leaves the cursor in the zoom box, TrackBox leaves the zoom box highlighted, as illustrated in Figure 4-20. If the user moves the cursor out of the zoom box, TrackBox removes the highlighting.

When the user releases the mouse button, TrackBox returns TRUE if the cursor is still in the zoom box and FALSE if it is not. If TrackBox returns TRUE, DoMouseDown calls the application-defined procedure DoZoomWindow to zoom the window. Listing 4-12 on page 4-55 shows the DoZoomWindow procedure.

Figure 4-20 The zoom box with and without highlighting



Handling Keyboard Events in Windows

Whenever your application is the foreground process, it receives key-down events for all keyboard activity, except for the three standard Command–Shift–number key sequences and any other Command–Shift–number key combinations the user has installed. (Command–Shift–1 and Command–Shift–2 eject disks, and Command–Shift–3 stores a snapshot of the screen in a TeachText document on the startup volume. Your application never receives these key combinations, which are handled by the Event Manager. For more information, see the chapter "Event Manager" in this book.)

In general, the active window is the target of keyboard activity.

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. To ensure consistent use of and response to keyboard events, follow the guidelines in *Macintosh Human Interface Guidelines*. Your application should, for example, allow the user to choose frequently used menu items by pressing a keyboard equivalent—usually a combination of the Command key and another key.

When you receive a key-down event, you first check whether the user is holding down a modifier key (Command, Shift, Control, Caps Lock, and Option, on a standard keyboard) and another key at the same time. If the Command key and a character key are held down simultaneously, for example, you adjust your menus, enabling and disabling items as appropriate, and allow the user to choose the menu item associated with the Command-key combination.

Typically, your application provides feedback for standard keystrokes by drawing the character on the screen. It should also recognize arrow keys for moving the cursor within a text display, and it might add support for function keys or other special keys available on nonstandard keyboards.

For an example of an application-defined routine for handling keyboard events, see the chapter "Event Manager" in this book.

Handling Update Events

The Event Manager sends your application an update event when part or all of your window's content region needs to be redrawn. Specifically, the Event Manager checks each window's update region every time your application calls `WaitNextEvent` or `EventAvail` (or `GetNextEvent`) and generates an update event for every window whose update region is not empty.

The Window Manager typically triggers update events when the moving and relayering of windows on the screen require that one or more windows be redrawn. If the user moves a window that covers part of an inactive window, for example, the Window Manager first calls the window definition function of the inactive window, requesting that it draw the window frame. It then adds the newly exposed area to the window's update region, which triggers an update event asking your application to update the content region. Your application can also trigger update events itself by manipulating the update region.

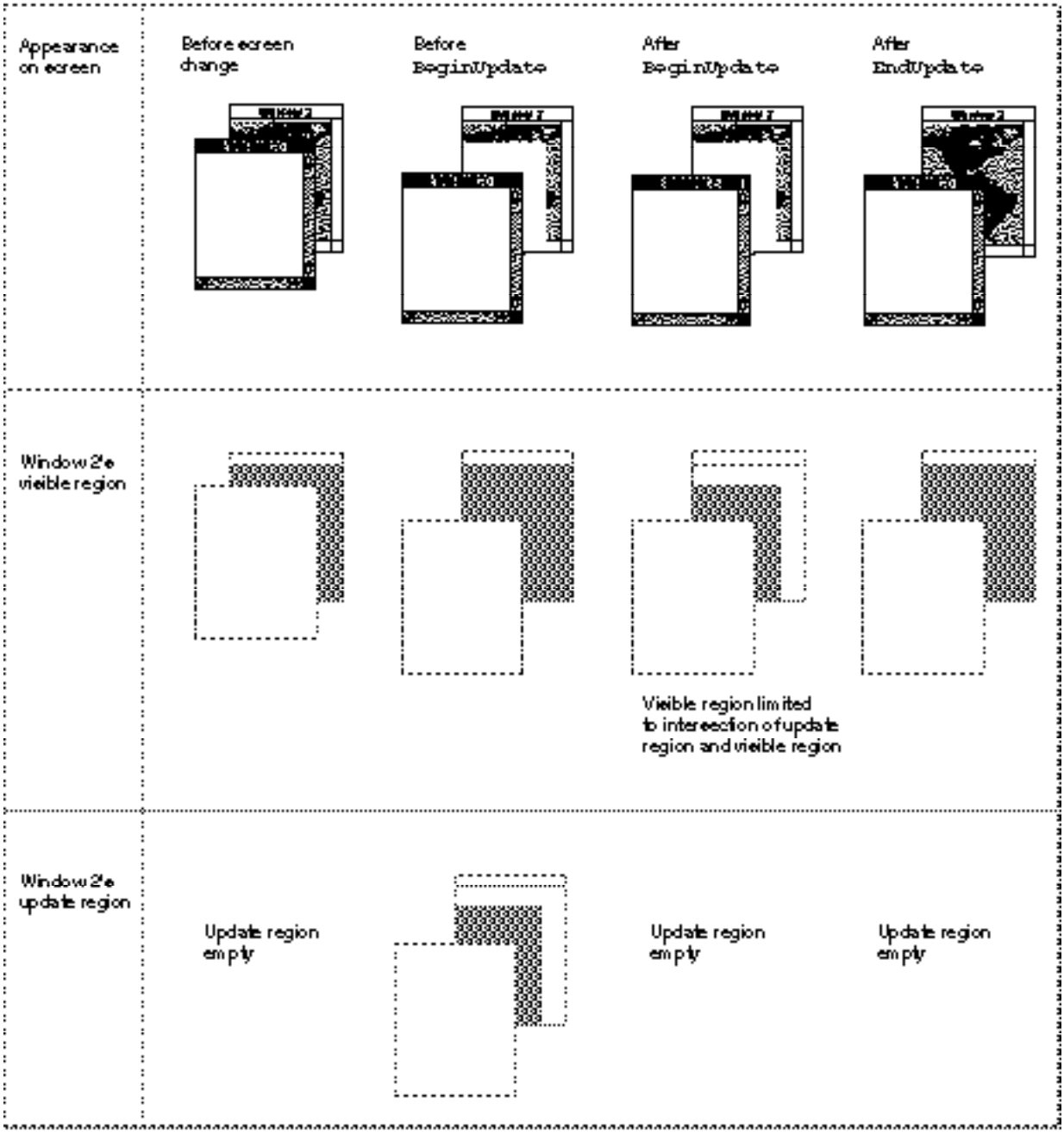
Your application can receive update events when it is in either the foreground or the background.

The Window Manager ensures that you do not accidentally draw in other windows by clipping all screen drawing to the visible region of a window's graphics port. The **visible region** is the part of the graphics port that's actually visible on the screen—that is, the part that's not covered by other windows. The Window Manager stores a handle to the visible region in the `visRgn` field of the graphics port data structure, which itself is in the window record.

In response to an update event, your application calls the `BeginUpdate` procedure, draws the window's contents, and then calls the `EndUpdate` procedure. As illustrated in Figure 4-21, `BeginUpdate` limits the visible region to the intersection of the visible region and the update region. Your application can then update either the visible region or the entire content region—because QuickDraw limits drawing to the visible region, only the parts of the window that actually need updating are drawn. The `BeginUpdate` procedure also clears the update region. After you've updated the window, you call `EndUpdate` to restore the visible region in the graphics port to the full visible region.

See *Inside Macintosh: Imaging* for more information about graphics ports and visible regions.

Figure 4-21 The effects of `BeginUpdate` and `EndUpdate` on the visible region and update region



Listing 4-10 illustrates an application-defined procedure, `DoUpdate`, that handles an update event.

Listing 4-10 Handling update events

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: LongInt;
BEGIN
   {determine type of window as defined by this application}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:           {document window}
         BEGIN
            BeginUpdate(window);
            MyDrawWindow(window);
            EndUpdate(window);
         END;
      OTHERWISE               {alert or dialog box}
         DoUpdateMyDialog(window);
      END; {of CASE}
END;
```

The `DoUpdate` procedure first determines whether the window being updated is a document window or some other application-defined window by calling the application-defined procedure `MyGetWindowType` (shown in Listing 4-1 on page 4-25). If the window is a document window, `DoUpdate` calls `BeginUpdate` to establish the temporary visible region, calls the application-defined procedure `MyDrawWindow` (shown in Listing 4-8 on page 4-39) to redraw the content region, and then calls `EndUpdate` to restore the visible region.

If the window is an alert box or a dialog box, `DoUpdate` calls the application-defined procedure `DoUpdateMyDialog`, which is not shown here.

Handling Activate Events

Your application activates and deactivates windows in response to **activate events,** which are generated by the Window Manager 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).

Your application often triggers activate events itself by calling the `SelectWindow` procedure. When it receives a mouse-down event in an inactive window, for example, your application calls `SelectWindow`, which brings the selected window to the front, removes the highlighting from the previously active window, and adds highlighting to the selected window. The `SelectWindow` procedure then generates two activate events: the first one tells your application to deactivate the previously active window; the second, to activate the newly active window.

When you receive the event for the previously active window, you

- hide the controls and size box
- remove or alter any highlighting of selections in the window

When you receive the event for the newly active window, you

- draw the controls and size box
- restore the content area as necessary, adding the insertion point in its former location or highlighting any previously highlighted selections

If the newly activated window also needs updating, your application also receives an update event, as described in the previous section, "Handling Update Events."

Note

A switch to one of your application's windows from a different application is handled through suspend and resume events, not activate events. See the chapter "Event Manager" in this book for a description of how your application can share processing time. ◆

Listing 4-11 illustrates the application-defined procedure `DoActivate`, which handles activate events.

Listing 4-11 Handling activate events

```
PROCEDURE DoActivate (window: WindowPtr; activate: Boolean;
                      event: EventRecord);
VAR
   windowType:               Integer;
   myData:                   MyDocRecHnd;
   growRect:                 Rect;
BEGIN
   {determine type of window as defined by this application}
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyFindModelessDialogBox:     {modeless Find dialog box}
         DoActivateFindDBox(window, event);
                        {modeless Check Spelling dialog box}
      kMyCheckSpellingModelessDialogBox:
         DoActivateCheckSpellDBox(window, event);
      kMyDocWindow:                 {document window}
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window)); {get document record}
            HLock(Handle(myData));  {lock document record}
            WITH myData^^ DO
            IF activate THEN        {window is becoming active}
```

```
            BEGIN
               {restore selections and insert caret--if using }
               { TextEdit, for example, call TEActivate}
               TEActivate(editRec);
               MyAdjustMenus;       {adjust menus for window}
                                    {handle the controls}
               docVScroll^^.contrlVis := kControlVisible;
               docHScroll^^.contrlVis := kControlVisible;
               InvalRect(docVScroll^^.contrlRect);
               InvalRect(docHScroll^^.contrlRect);
               growRect := window^.portRect;
               WITH growRect DO     {handle the size box}
                  BEGIN             {adjust for the scroll bars}
                     top := bottom - kScrollbarAdjust;
                     left := right - kScrollbarAdjust;
                  END;
               InvalRect(growRect);
            END
            ELSE           {window is becoming inactive}
            BEGIN
               TEDeactivate(editRec);     {call TextEdit to deactivate data}
               HideControl(docVScroll);   {hide the scroll bars}
               HideControl(docHScroll);
               DrawGrowIcon(window);      {draw the size box}
            END;
         HUnLock(Handle(myData));         {unlock document record}
      END; {of kMyDocWindow statement}
   END; {of CASE statement}
END;
```

The `DoActivate` procedure first determines the general type of the window; that is, it calls an application-defined function that returns a constant identifying the type of the window: a Find dialog box, a Check Spelling dialog box, or a document window. Listing 4-1 on page 4-25 shows the `MyGetWindowType` function.

If the target of the activate event is a dialog box window, `DoActivate` calls other application-defined routines for activating and deactivating those dialog boxes. The `DoActivateFindDBox` and `DoActivateCheckSpellDBox` routines are not shown here. (The `DoActivate` procedure does not check for alert boxes and modal dialog boxes, because the Dialog Manager's `ModalDialog` procedure automatically handles activate events.)

If the target is a document window and the activate event specifies that the window is becoming active, `DoActivate` highlights any user selections in the window and draws the insertion point where appropriate. It then makes the controls visible, adds the area occupied by the scroll bars to the update region, and adds the area occupied by the size box to the update region. (Placing window area in the update region guarantees an update event. When the application receives the update event, it calls the application-defined procedure `DoUpdate` to draw the update region, which in this case includes the size box and scroll bars.)

If the target is a document window, and the activate event specifies that the window is becoming inactive, the `DoActivate` procedure calls the TextEdit procedure `TEDeactivate` to remove highlighting from user selections, calls the Control Manager procedure `HideControl` to hide the scroll bars, and calls the Window Manager procedure `DrawGrowIcon` to draw the size box and the outline of the scroll bar area.

Moving a Window

When the user drags a window by the title bar (except for the close and zoom box regions), the window should move, following the cursor as it moves on the desktop. Your application can easily let the user move the window by calling the `DragWindow` procedure.

The `DragWindow` procedure draws an outline of the window on the screen and moves the outline as the user moves the mouse. When the user releases the mouse button, `DragWindow` calls the `MoveWindow` function, which redraws the window in its new location.

For an example of moving a window, see the `inDrag` case in Listing 4-9 on page 4-44.

Zooming a Window

The zoom box allows the user to alternate quickly between two window positions and sizes: the user state and the standard state.

The **user state** is the window size and location established by the user. If your application does not supply an initial user state, the user state is simply the size and location of the window when it was created, until the user resizes it.

The **standard state** is the window size and location that your application considers most convenient, considering the function of the document and the screen space available. In a word-processing application, for example, a standard-state window might show a full page, if possible, or a page of full width and as much length as fits on the screen. If the user changes the page size through Page Setup, the application might adjust the standard state to reflect the new page size. If your application does not define a standard state, the Window Manager automatically sets the standard state to the entire gray region on the main screen, minus a three-pixel border on all sides. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.) The user cannot change a window's standard state.

The user and standard states are stored in a record whose handle appears in the `dataHandle` field of the window record.

```
TYPE WStateData =
   RECORD
      userState:  Rect;    {size and location established by user}
      stdState:   Rect;    {size and location established by }
                           { application}
   END;
```

The Window Manager sets the initial values of the `userState` and `stdState` fields when it fills in the window record, and it updates the `userState` field whenever the user resizes the window. You typically compute the standard state every time the user zooms to the standard state, to ensure that you're zooming to an appropriate location.

When the user presses the mouse button with the cursor in the zoom box, the `FindWindow` function specifies whether the window is in the user state or the standard state: when the window is in the standard state, `FindWindow` returns `inZoomIn` (meaning that the window is to be zoomed "in" to the user state); when the window is in the user state, `FindWindow` returns `inZoomOut` (meaning that the window is to be zoomed "out" to the standard state).

When `FindWindow` returns either `inZoomIn` or `inZoomOut`, your application can call the `TrackBox` function to handle the highlighting of the zoom box and to determine whether the cursor is inside or outside the box when the button is released. If `TrackBox` returns `TRUE`, your application can call the `ZoomWindow` procedure to resize the window (after computing a new standard state). If `TrackBox` returns `FALSE`, your application doesn't need to do anything. Listing 4-9 on page 4-44 illustrates the use of `TrackBox` in an event-handling routine.

Listing 4-12 illustrates an application-defined procedure, `DoZoomWindow`, which an application might call when `TrackBox` returns `TRUE` after `FindWindow` returns either `inZoomIn` or `inZoomOut`. Because the user might have moved the window to a different screen since it was last zoomed, the procedure first determines which screen contains the largest area of the window and then calculates the ideal window size for that screen before zooming the window.

The screen calculations in the `DoZoomWindow` procedure depend on the routines for handling graphics devices that were introduced at the same time as Color QuickDraw. Therefore, `DoZoomWindow` checks for the presence of Color QuickDraw before comparing the window to be zoomed with the graphics devices in the device list. If Color QuickDraw is not available, `DoZoomWindow` assumes that it's running on a computer with a single screen.

Listing 4-12 Zooming a window

```
PROCEDURE DoZoomWindow (thisWindow: windowPtr; zoomInOrOut: Integer);
VAR
   gdNthDevice, gdZoomOnThisDevice: GDHandle;
   savePort:                        GrafPtr;
   windRect, zoomRect, theSect:     Rect;
   sectArea, greatestArea:          LongInt;
   wTitleHeight:                    Integer;
   sectFlag:                        Boolean;
BEGIN
   GetPort(savePort);
   SetPort(thisWindow);
   EraseRect(thisWindow^.portRect);    {erase to avoid flicker}
   IF zoomInOrOut = inZoomOut THEN     {zooming to standard state}
   BEGIN
      IF NOT gColorQDAvailable THEN    {assume a single screen and }
      BEGIN                            { set standard state to full screen}
         zoomRect := screenBits.bounds;
         InsetRect(zoomRect, 4, 4);
         WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                              := zoomRect;
      END
      ELSE                    {locate window on available graphics devices}
      BEGIN
         windRect := thisWindow^.portRect;
         LocalToGlobal(windRect.topLeft);    {convert to global coordinates}
         LocalToGlobal(windRect.botRight);
         {calculate height of window's title bar}
         wTitleHeight := windRect.top - 1 -
                     WindowPeek(thisWindow)^.strucRgn^^.rgnBBox.top;
         windRect.top := windRect.top - wTitleHeight;
         gdNthDevice := GetDeviceList;
         greatestArea := 0;         {initialize to 0}
         {check window against all gdRects in gDevice list and remember }
         { which gdRect contains largest area of window}
         WHILE gdNthDevice <> NIL DO
         IF TestDeviceAttribute(gdNthDevice, screenDevice) THEN
            IF TestDeviceAttribute(gdNthDevice, screenActive) THEN
            BEGIN
               {The SectRect routine calculates the intersection }
               { of the window rectangle and this gDevice }
               { rectangle and returns TRUE if the rectangles intersect, }
               { FALSE if they don't.}
```

```
            sectFlag := SectRect(windRect, gdNthDevice^^.gdRect,
                                 theSect);
            {determine which screen holds greatest window area}
            {first, calculate area of rectangle on current device}
            WITH theSect DO
               sectArea := LongInt(right - left) * (bottom - top);
            IF sectArea > greatestArea THEN
            BEGIN
               greatestArea := sectArea;  {set greatest area so far}
               gdZoomOnThisDevice := gdNthDevice;  {set zoom device}
            END;
            gdNthDevice := GetNextDevice(gdNthDevice);
         END;  {of WHILE}
      {if gdZoomOnThisDevice is on main device, allow for menu bar height}
      IF gdZoomOnThisDevice = GetMainDevice THEN
         wTitleHeight := wTitleHeight + GetMBarHeight;
      WITH gdZoomOnThisDevice^^.gdRect DO    {create the zoom rectangle}
      BEGIN
         {set the zoom rectangle to the full screen, minus window title }
         { height (and menu bar height if necessary), inset by 3 pixels}
         SetRect(zoomRect, left + 3, top + wTitleHeight + 3,
                right - 3, bottom - 3);
         {If your application has a different "most useful" standard }
         { state, then size the zoom window accordingly.}
         {set up the WStateData record for this window}
         WStateDataHandle(WindowPeek(thisWindow)^.dataHandle)^^.stdState
                                                          := zoomRect;
      END;
   END;
END; {of inZoomOut}
{if zoomInOrOut = inZoomIn, just let ZoomWindow zoom to user state}
{zoom the window frame}
ZoomWindow(thisWindow, zoomInOrOut, (thisWindow = FrontWindow));
MyResizeWindow(thisWindow);   {application-defined window-sizing routine}
SetPort(savePort);
END; (of DoZoomWindow)
```

If the user is zooming the window to the standard state, `DoZoomWindow` calculates a new standard size and location based on the application's own considerations, the current location of the window, and the available screens. The `DoZoomWindow` procedure always places the standard state on the screen where the window is currently displayed or, if the window spans screens, on the screen containing the largest area of the window.

The bulk of the code in Listing 4-12 is devoted to determining which screen should display the window in the standard state. The sample code shown here establishes a standard state that simply occupies the gray area on the chosen screen, minus three pixels on all sides. Your application should establish a standard state appropriate to its own documents. When calculating the standard state, move the window as little as possible from the user state. If possible, anchor one corner of the standard state rectangle to one corner of the user state rectangle.

If the user is zooming the window to the user state, `DoZoomWindow` doesn't have to perform any calculations, because the user state rectangle stored in the state data record should represent a valid screen location.

After calculating the standard state, if necessary, `DoZoomWindow` calls the `ZoomWindow` procedure to redraw the window frame in the new size and location and then calls the application-defined procedure `MyResizeWindow` to redraw the window's content region. Listing 4-14 on page 4-59 shows the `MyResizeWindow` procedure.

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 `GrowWindow` function. This function displays a **grow image**—a gray 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.

To avoid unmanageably large or small windows, you supply lower and upper size limits when you call `GrowWindow`. The `sizeRect` parameter to `GrowWindow` specifies both the lower and upper size limits in a single structure of type `Rect`. The values in the `sizeRect` structure represent window dimensions, not screen coordinates:

- You supply the minimum vertical measurement in `sizeRect.top`.
- You supply the minimum horizontal measurement in `sizeRect.left`.
- You supply the maximum vertical measurement in `sizeRect.bottom`.
- You supply the maximum horizontal measurement in `sizeRect.right`.

Most applications specify a minimum size big enough to include all parts of the structure area and the scroll bars. Because the user cannot move the cursor beyond the edges of the screen, you can safely set the maximum size to the largest possible rectangle.

When the user releases the mouse button, `GrowWindow` returns a long integer that describes the window's new height (in the high-order word) and width (in the low-order word). A value of 0 means that the window's size did not change. When `GrowWindow` returns any value other than 0, you call `SizeWindow` to resize the window.

Note
Use the utility functions `HiWord` and `LoWord` to retrieve the high-order and low-order words, respectively. ◆

When you change a window's size, you must erase and redraw the window's scroll bars.

Listing 4-13 illustrates the application-defined procedure `DoGrowWindow` for tracking mouse activity in the size box and resizing the window.

Listing 4-13 Resizing a window

```
PROCEDURE DoGrowWindow (thisWindow: windowPtr;
                        event: EventRecord);
VAR
   growSize:         LongInt;
   limitRect:        Rect;
   oldViewRect:      Rect;
   locUpdateRgn:     RgnHandle;
   theResult:        Boolean;
   myData:           MyDocRecHnd;
BEGIN
   {set up the limiting rectangle: kMinDocSize = 64 }
                                { kMaxDocSize = 65535}
   SetRect(limitRect, kMinDocSize, kMinDocSize, kMaxDocSize,
            kMaxDocSize);
   {call Window Manager to let user drag size box}
   growSize := GrowWindow(thisWindow, event.where, limitRect);
   IF growSize <> 0 THEN         {if user changed size, }
   BEGIN                         { then resize window}
      myData := MyDocRecHnd(GetWRefCon(thisWindow));
      oldViewRect := myData^^.editRec^^.viewRect;
      locUpdateRgn := NewRgn;
      {save update region in local coordinates}
      MyGetLocalUpdateRgn(thisWindow, locUpdateRgn);
      {resize the window}
      SizeWindow(thisWindow, LoWord(growSize), HiWord(growSize),
                  TRUE);
      MyResizeWindow(thisWindow);
      {find intersection of old viewRect and new viewRect}
      theResult := SectRect(oldViewRect,
                            myData^^.editRec^^.viewRect,
                            oldViewRect);
      {validate the intersection (don't update)}
      ValidRect(oldViewRect);
```

```
      {invalidate any prior update region}
      InvalRgn(locUpdateRgn);
      DisposeRgn(locUpdateRgn);
   END;
END;
```

When the user presses the mouse button while the cursor is in the size box, the procedure that handles mouse-down events (`DoMouseDown`, shown on page 4-44) calls the application-defined `DoGrowWindow` procedure. The `DoGrowWindow` procedure calls the Window Manager function `GrowWindow`, which tracks mouse movement as long as the button is held down. If the user drags the size box before releasing the mouse button, `GrowWindow` returns a nonzero value, and `DoGrowWindow` prepares to resize the window. First `DoGrowWindow` saves the current view rectangle in the variable `oldViewRect`. It will use this information later, when redrawing the content region of the window in its new size. The `GrowWindow` procedure also saves the current update region, in local coordinates, in the region `LocUpdateRgn`, so that it can restore the update region after doing its own update-region maintenance. (This step is necessary only if an application allows user input to accumulate into the update region, drawing in response to update events instead of drawing into the window immediately.)

After saving the current view rectangle and the current update region, `DoGrowWindow` calls the Window Manager procedure `SizeWindow` to draw the window in its new size. The `DoGrowWindow` procedure then calls the application-defined procedure `MyResizeWindow`, which adjusts the window scroll bars and window contents to the new size. Listing 4-14 illustrates the application-defined `MyResizeWindow` procedure.

After calling `SizeWindow`, `DoGrowWindow` calculates the intersection of the old view rectangle and the new view rectangle. It uses this area to revalidate unchanged portions of the window (that is, to remove them from the update region), because the `MyResizeWindow` procedure invalidates the entire window (that is, places the entire window in the update region). This way, only the changed parts of the content area are redrawn when the application receives its next update event.

Listing 4-14 Adjusting scroll bars and content region when resizing a window

```
PROCEDURE MyResizeWindow (window: WindowPtr);
BEGIN
   WITH window^ DO
   BEGIN
      {adjust scroll bars and contents-- }
      { see the chapter “Control Manager” for implementation}
      MyAdjustScrollbars(window, TRUE);
      MyAdjustTE(window);
      {invalidate content region, forcing an update}
      InvalRect(portRect);
   END;
END; {MyResizeWindow}
```

Listing 4-15 illustrates the application-defined procedure `MyGetLocalUpdateRgn`, which supplies a window's update region in local coordinates. The `MyGetLocalUpdateRgn` procedure uses the QuickDraw routines `CopyRgn` and `OffsetRgn`, documented in *Inside Macintosh: Imaging*.

Listing 4-15 Converting a window region to local coordinates

```
PROCEDURE MyGetLocalUpdateRgn (window: WindowPtr;
                               localRgn: RgnHandle);
BEGIN
   {save old update region}
   CopyRgn(WindowPeek(window)^.updateRgn, localRgn);
   WITH window^.portBits.bounds DO
      OffsetRgn(localRgn, left, top);  {convert to local coords}
END; {MyGetLocalUpdateRgn}
```

Closing a Window

The user closes a window either by clicking the close box, in the upper-left corner of the window, or by choosing Close from the File menu.

When the user presses the mouse button while the cursor is in the close box, your application calls the `TrackGoAway` function to track the mouse until the user releases the button, as illustrated in Listing 4-9 on page 4-44. If the user releases the button while the cursor is outside the close box, `TrackGoAway` returns `FALSE`, and your application does nothing. If `TrackGoAway` returns `TRUE`, your application invokes its own procedure for closing a window.

The specific steps you take when closing a window depend on what kind of information the window contains and whether the contents need to be saved. The sample code in this chapter recognizes four kinds of windows: the modeless dialog box containing the Find dialog, the modeless dialog box containing the Spell Check dialog, a standard document window, and a window associated with a desk accessory that was launched in the application's partition.

Listing 4-16 illustrates an application-defined procedure, `DoCloseCmd`, that determines what kind of window is being closed and follows the appropriate strategy. The application calls `DoCloseCmd` when the user clicks a window's close box or chooses Close from the File menu.

Listing 4-16 Handling a close command

```
PROCEDURE DoCloseCmd;
VAR
   myWindow:   WindowPtr;     {pointer to window's record}
   myData:     MyDocRecHnd;   {handle to a document record}
   windowType: Integer;       {application-defined window type}
```

```
BEGIN
   myWindow := FrontWindow;
   windowType := MyGetWindowType(myWindow);
   CASE windowType OF
      kMyFindModelessDialog:        {for modeless dialog boxes, }
         HideWindow(myWindow);      { hide window}
      kMySpellModelessDialog:       {for modeless dialog boxes, }
         HideWindow(myWindow);      { hide window}
      kDAWindow:              {for desk accessories, close the DA}
         CloseDeskAcc(WindowPeek(myWindow)^.windowKind);
      kMyDocWindow:           {for documents, handle file first}
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(myWindow));
            MyCloseDocument(myData);
         END;
   END;     {of CASE}
END;
```

The `DoCloseCmd` procedure first determines which window is the active window and then calls the application-defined function `MyGetWindowType` to identify the window's type, as defined by the application. If the window is a modeless dialog box, `MyCloseCmd` merely hides the window, leaving the data structures in memory. For a sample routine that displays a hidden window, see Listing 4-18 on page 4-64.

If the window is associated with a desk accessory, the `DoCloseCmd` procedure calls the `CloseDeskAcc` procedure to close the desk accessory. This case is included only for compatibility; in System 7 desk accessories are seldom launched in an application's partition.

If the window is associated with a document, `DoCloseCmd` reads the document record and then calls the application-defined procedure `MyCloseDocument` to handle the closing of a document window. Listing 4-17 illustrates the `MyCloseDocument` procedure.

Listing 4-17 Closing a document

```
PROCEDURE MyCloseDocument (myData: MyDocRecHnd);
VAR
   title:      Str255;        {window/document title}
   item:       Integer;       {item in Save Alert dialog box}
   docWindow:  WindowPtr;     {pointer to window record}
   event:      EventRecord;   {dummy record for DoActivate}
   myErr:      OSErr;         {variable for error-checking}
BEGIN
   docWindow := FrontWindow;
   IF (myData^^.windowDirty) THEN   {changed since last save}
```

```
   BEGIN
      GetWTitle(docWindow, title);     {get window title}
      ParamText(title, '', '', '');    {set up dialog text}
      {deactivate window before displaying Save dialog}
      DoActivate(docWindow, FALSE, event);
      {put up Save dialog and retrieve user response}
      item := CautionAlert(kSaveAlertID, @MyEventFilter);
      IF item = kCancel THEN     {user clicked Cancel}
         Exit(MyCloseDocument);  {exit without closing}
      IF item = kSave THEN       {user clicked Save}
         DoSaveCmd;              {save the document}
      {otherwise user clicked Don't Save-- }
      { close document in either case}
      myErr := DoCloseFile(myData); {close document}
      {Add your own error handling.}
   END;
   {close window whether or not user saved}
   CloseWindow(docWindow);          {close window}
   DisposePtr(Ptr(docWindow));      {dispose of window record}
END;
```

The `MyCloseDocument` procedure checks the `windowDirty` field in the document record (described in "Managing Multiple Windows" beginning on page 4-23). If the value of `windowDirty` is `TRUE`, `MyCloseDocument` displays a dialog box giving the user a chance to save the document before closing the window. The dialog box gives the user the choices of canceling the close, saving the document before closing the window, or closing the window without saving the document. If the user cancels, `MyCloseDocument` merely exits. If the user opts to save the document, `MyCloseDocument` calls the application-defined routine `DoSaveCmd`, which is not shown here. (For a description of how to save and close a file, see the chapter "Introduction to File Management" in *Inside Macintosh: Files*.) Whether or not the user saves the document before closing the window, `MyCloseDocument` closes the document and finally removes the window from the screen and diposes of the memory allocated to the window record.

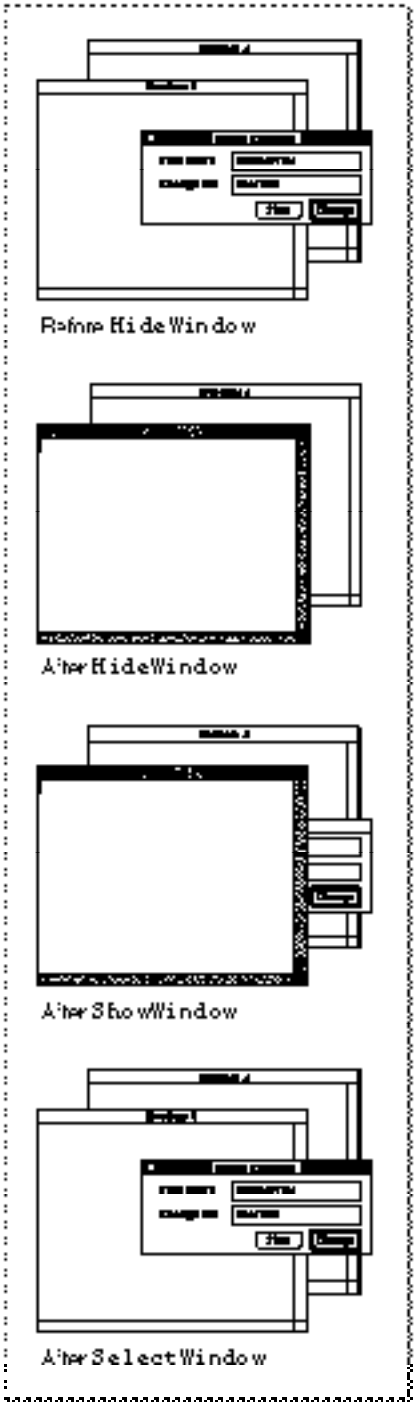
Hiding and Showing a Window

Whenever the user clicks a window's close box, you remove the window from the screen. Sometimes, however, you might find it's more efficient to merely hide the window, instead of removing its data structures.

If your application includes a Find modeless dialog box that searches for a string, for example, you might want to keep the structures in memory as long as the user is working. When the user closes the dialog box by clicking the close box, you simply hide the window by calling the `HideWindow` procedure. The next time the user chooses the Find command, your dialog box window is already available, in the same location and with the same text selected as when it was last used.

To reverse the `HideWindow` procedure, you must call both `ShowWindow`, which makes the window visible, and `SelectWindow`, which makes it the active window. Figure 4-22 illustrates how the three procedures affect the window's status on the screen.

Figure 4-22 The cumulative effects of `HideWindow`, `ShowWindow`, and `SelectWindow`



The application-defined procedure for closing a window—DoCloseCmd, described on page 4-60—hides the Find and Spell Check dialog box windows when the user closes them. Listing 4-18 illustrates a sample application-defined procedure, DoShowModelessFindDialogBox, for redisplaying the Find dialog box when the user next chooses the Find command.

Listing 4-18 Showing a hidden dialog box

```
PROCEDURE DoShowModelessFindDialogBox;
BEGIN
   IF gFindDialog = NIL THEN     {no Find dialog box exists yet}
   BEGIN
      {create Find dialog box}
      gFindDialog := GetNewDialog(rFindModelessDialog, NIL,
                                  Pointer(-1));
      IF gFindDialog = NIL THEN        {creation failed}
         Exit(DoShowModelessFindDialogBox);     {exit}
      {store value that identifies dbox in window refCon field}
      SetWRefCon(gFindDialog, LongInt(kMyFindModelessDialog))
      ShowWindow(gFindDialog);         {make dialog box visible}
   END
   ELSE              {dialog box already exists}
   BEGIN
      ShowWindow(gFindDialog);         {make it visible}
      SelectWindow(gFindDialog);       {select it}
   END;
END;
```

The DoShowModelessFindDialogBox procedure first checks whether the Find dialog box already exists. If it doesn't, then DoShowModelessFindDialogBox creates a new dialog box through the Dialog Manager. It stores the constant that represents the Find dialog box in the refCon field of the new window record, makes the window visible, and draws the dialog box contents. If the Find dialog box already exists, DoShowModelessFindDialogBox makes the dialog box window visible and selects it. When the Window Manager then generates an activate event, the application calls its own procedure to draw the contents.

Window Manager Reference

This section describes the Window Manager's data structures and routines. It also lists the resources used by the Window Manager and describes the window ('WIND') and window color table ('wctb') resources.

Data Structures

This section describes the Window Manager data structures: the window record, the color window record, the state data record, the window color table record, the auxiliary window record, and the window list.

A window record or color window record describes an individual window. It includes the record for the graphics port in which the window is displayed.

The state data record stores two rectangles, known as the user state and the standard state, which define the size and location of the window as specified by the user and by your application. Your application switches between the two states when the user clicks the zoom box.

A window color table defines the colors to be used for drawing the window's frame and highlighting selected text. Ordinarily, you use the default window color table, which produces windows in the colors selected by the user through the Color control panel. If your application has some unusual need to control the frame colors, you can set up your own window color tables.

The Window Manager uses auxiliary window records to associate a window with its window color table.

The Window Manager uses the window list to track all of the windows on the desktop.

The Color Window Record

The Window Manager maintains a window record or color window record for each window on the desktop.

The Window Manager supplies routines that let you access the window record as necessary. Your application seldom changes fields in the window record directly.

The `CWindowRecord` data type defines the window record for a color window. The `CWindowPeek` data type is a pointer to a color window record. The first field in the window record is in fact the record that describes the window's graphics port. The `CWindowPtr` data type is defined as a pointer to the window's graphics port.

When Color QuickDraw is not available, you can create monochrome windows using the parallel data types `WindowRecord`, `WindowPeek`, and `WindowPtr`, described in the next section, "The Window Record."

For compatibility, the `WindowPtr` and `WindowPeek` data types can point to either a color window record or a monochrome window record. You use the `WindowPtr` data type to specify a window in most Window Manager routines, and you can use it to specify a graphics port in QuickDraw routines that take the `GrafPtr` data type. Note that you can access only the fields of the window's graphics port, not the rest of the window record, through the `WindowPtr` and `CWindowPtr` data types. You use the `WindowPeek` and `CWindowPeek` data types in low-level Window Manager routines and in your own routines that access window record fields beyond the graphics port.

The routines that manipulate color windows get color information from the window color tables and the auxiliary window record described in the sections "The Window Color Table Record" on page 4-71 and "The Auxiliary Window Record" on page 4-73.

```
TYPE  CWindowPtr  = ^CGrafPtr;
      CWindowPeek = ^CWindowRecord;

TYPE  CWindowRecord =
   RECORD
      port:          CGrafPort;     {window's graphics port}
      windowKind:    Integer;       {class of the window}
      visible:       Boolean;       {visibility}
      hilited:       Boolean;       {highlighting}
      goAwayFlag:    Boolean;       {presence of close box}
      spareFlag:     Boolean;       {presence of zoom box}
      strucRgn:      RgnHandle;     {handle to structure region}
      contRgn:       RgnHandle;     {handle to content region}
      updateRgn:     RgnHandle;     {handle to update region}
      windowDefProc: Handle;        {handle to window definition }
                                    { function}
      dataHandle:    Handle;        {handle to window state }
                                    { data record}
      titleHandle:   StringHandle;  {handle to window title}
      titleWidth:    Integer;       {title width in pixels}
      controlList:   ControlHandle; {handle to control list}
      nextWindow:    CWindowPeek;   {pointer to next window }
                                    { record in window list}
      windowPic:     PicHandle;     {handle to optional picture}
      refCon:        LongInt;       {storage available to your }
                                    { application}
   END;
```

Field descriptions

`port` The graphics port record that describes the graphics port in which the window is drawn.

The graphics port record, which is documented in *Inside Macintosh: Imaging*, defines the rectangle in which drawing can occur, the window's visible region, the window's clipping region, and a collection of current drawing characteristics such as fill pattern, pen location, and pen size.

`windowKind` The class of window—that is, how the window was created.

The Window Manager fills in this field when it creates the window record. It places a negative value in `windowKind` when the window

was created by a desk accessory. (The value is the reference ID of the desk accessory.) This field can also contain one of two constants:

```
CONST
   dialogKind  = 2;     {dialog or alert window}
   userKind    = 8;     {window created by an }
                        { application}
```

The value `dialogKind` identifies all dialog or alert box windows, whether created by the system software or, indirectly through the Dialog Manager, by your application. The Dialog Manager uses this field to help it track dialog and alert box windows.

The value `userKind` represents a window created directly by your application.

`visible` A Boolean value indicating whether or not the window is visible. If the window is visible, the Window Manager sets this field to `TRUE`; if not, `FALSE`. Visibility means only whether or not the window is to be displayed, not necessarily whether you can see it on the screen. (For example, a window that is completely covered by other windows can still be visible, even if the user cannot see it on the screen.)

`hilited` A Boolean value indicating whether the window is highlighted—that is, drawn with stripes in the title bar. Only the active window is ordinarily highlighted. When the window is highlighted, the `hilited` field contains `TRUE`; when not, `FALSE`.

`goAwayFlag` A Boolean value indicating whether the window has a close box.

The Window Manager fills in this field when it creates the window according to the information in the `'WIND'` resource or the parameters passed to the function that creates the window.

If the value of `goAwayFlag` is `TRUE`, and if the window type supports a close box, the Window Manager draws a close box when the window is highlighted.

`spareFlag` A Boolean value indicating whether the window type supports zooming. The Window Manager sets this field to `TRUE` if the window's type is one that includes a zoom box (`zoomDocProc`, `zoomNoGrow`, or even `modalDBoxProc + zoomDocProc`).

`strucRgn` A handle to the structure region, which is defined in global coordinates. The structure region is the entire screen area covered by the window—that is, both the window contents and the window frame.

`contRgn` A handle to the content region, which is defined in global coordinates. The content region is the part of the window that contains the document, dialog, or other data; the window controls; and the size box.

`updateRgn` A handle to the update region, which is defined in global coordinates. The update region is the portion of the window that must be redrawn. It is maintained jointly by the Window Manager and your application. The update region excludes parts of the window that are covered by other windows.

`windowDefProc` A handle to the definition function that controls the window.

There's no need for your application to access this field directly.

In Macintosh models that use only 24-bit addressing, this field contains both a handle to the window's definition function and the window's variation code. If you need to know the variation code, regardless of the addressing mode, call the `GetWVariant` function.

`dataHandle` Usually a handle to a data area used by the window definition function.

For zoomable windows, `dataHandle` contains a handle to the `WStateData` record, which contains the user state and standard state rectangles. The `WStateData` record is described in "The Window State Data Record" beginning on page 4-70.

A window definition function that needs only 4 bytes of data can use the `dataHandle` field directly, instead of storing a handle to the data. The window definition function that handles rounded-corner windows, for example, stores the diameters of curvature in the `dataHandle` field.

`titleHandle` A handle to the string that defines the title of the window.

`titleWidth` The width, in pixels, of the window's title.

`controlList` A handle to the window's control list, which is used by the Control Manager. (See the chapter "Control Manager" in this book for a description of control lists.)

`nextWindow` A pointer to the next window in the window list, that is, the window behind this window on the desktop. In the window record for the last window on the desktop, the `nextWindow` field is set to `NIL`.

`windowPic` A handle to a QuickDraw picture of the window's contents. The Window Manager initially sets the `windowPic` field to `NIL`. If you're using the window to display a stable image, you can use the `SetWindowPic` procedure to place a handle to the picture in this field. When the window's contents need updating, the Window Manager then redraws the contents itself instead of generating an update event.

`refCon` The window's reference value field, which is simply storage space available to your application for any purpose. The sample code in this chapter uses the `refCon` field to associate a window with the data it displays by storing a window type constant in the `refCon` field of alert and dialog window records and a handle to a document record in the `refCon` field of a document window record.

Note

The close box, drag region, zoom box, and size box are not included in the window record because they don't necessarily have the formal data structure for regions as defined in QuickDraw. The window definition function determines where these regions are. ◆

The Window Record

If Color QuickDraw is not available, you create windows with a parallel data structure, the window record. The only difference between a color window record and a window record is that a color window record points to a color graphics port, which allows full use of Macintosh computers with color capability, and a window record points to a monochrome graphics port

The data types that describe window records, `WindowRecord`, `WindowPtr`, and `WindowPeek`, are parallel to the data types that describe color window records, and the fields in the monochrome window record are identical to the fields in the color window record. For a complete description, see "The Color Window Record" beginning on page 4-65.

```
TYPE  WindowPtr   = ^GrafPtr;
      WindowPeek  = ^WindowRecord;

TYPE  WindowRecord =                {all fields have same use }
   RECORD                           { as in color window record}
      port:          GrafPort;      {window's graphics port}
      windowKind:    Integer;       {class of the window}
      visible:       Boolean;       {visibility}
      hilited:       Boolean;       {highlighting}
      goAwayFlag:    Boolean;       {presence of close box}
      spareFlag:     Boolean;       {presence of zoom box}
      strucRgn:      RgnHandle;     {handle to structure region}
      contRgn:       RgnHandle;     {handle to content region}
      updateRgn:     RgnHandle;     {handle to update region}
      windowDefProc: Handle;        {handle to window definition }
                                    { function}
      dataHandle:    Handle;        {handle to window state }
                                    { data record}
      titleHandle:   StringHandle;  {handle to window title}
      titleWidth:    Integer;       {title width in pixels}
      controlList:   ControlHandle; {handle to control list}
      nextWindow:    WindowPeek;    {pointer to next window }
                                    { record in window list}
      windowPic:     PicHandle;     {handle to optional picture}
      refCon:        LongInt;       {storage available to your }
                                    { application}
   END;
```

The Window State Data Record

The zoom box allows the user to alternate quickly between two window positions and sizes: the user state and the standard state. The Window Manager stores the user state and your application stores the standard state in the window state data record, whose handle appears in the `dataHandle` field of the window record.

The `WStateData` record data type defines the window state data record.

```
TYPE  WStateDataPtr = ^WStateData;
      WStateDataHandle = ^WStateDataPtr;

      WStateData =
      RECORD
         userState:  Rect; {size and location established by user}
         stdState:   Rect; {size and location established by app}
      END;
```

Field descriptions

`userState` A rectangle that describes the window size and location established by the user.

The Window Manager initializes the user state to the size and location of the window when it is first displayed, and then updates the `userState` field whenever the user resizes a window. Although the user state specifies both the size and location of the window, the Window Manager updates the state data record only when the user resizes a window—not when the user merely moves a window.

`stdState` The rectangle describing the window size and location that your application considers the most convenient, considering the function of the document, the screen space available, and the position of the window in its user state. If your application does not define a standard state, the Window Manager automatically sets the standard state to the entire gray region on the main screen, minus a three-pixel border on all sides. The user cannot change a window's standard state.

Your application typically calculates and sets the standard state each time the user zooms to the standard state. In a word-processing application, for example, a standard state window might show a full page, if possible, or a page of full width and as much length as fits on the screen. If the user changes the page size through Page Setup, the application might adjust the standard state to reflect the new page size. (See *Macintosh Human Interface Guidelines* for a detailed description of how your application determines where to open and zoom windows.)

The `ZoomWindow` procedure changes the size of a window according to the values in the window state data record. The procedure changes the window to the user state when the user zooms "in" and to the standard state when the user zooms "out." For a detailed

description of zooming windows, see “Zooming a Window” beginning on page 4-53. For descriptions of the routines you call when zooming windows, see “Zooming Windows” beginning on page 4-101.

The Window Color Table Record

The user controls the colors used for the window frame and text highlighting through the Color control panel. Ordinarily, your application doesn’t override the user’s color choices, which are stored in a default window color table. If you have some extraordinary need to control window colors, you can do so by defining window color tables for your application’s windows.

The Window Manager maintains window color information tables in a data structure of type `WinCTab`.

You can define your own window color table and apply it to an existing window through the `SetWinColor` procedure.

To establish the window color table for a window when you create it, you provide a window color table (`'wctb'`) resource with the same resource ID as the `'WIND'` resource that defines the window.

The `WCTabPtr` data type is a pointer to a window color table record, and the `WTabHandle` is a handle to a window color table record.

```
TYPE  WCTabPtr = ^WinCTab;
      WCTabHandle = ^WCTabPtr;
```

The `WinCTab` data type defines a window color table record.

```
TYPE  WinCTab =
      RECORD
         wCSeed:     LongInt;    {reserved}
         wCReserved: Integer;    {reserved}
         ctSize:     Integer;    {number of entries in table -1}
         ctTable:    ARRAY[0..4] OF ColorSpec;
                                 {array of color specification }
                                 { records}
      END;
```

Field descriptions

`wCSeed`	Reserved.
`wCReserved`	Reserved.
`ctSize`	The number of entries in the table, minus 1. If you’re building a color table for use with the standard window definition function, the maximum value of this field is 12. Custom window definition functions can use color tables of any size.

`ctTable` An array of `colorSpec` records.

In a window color table, each `colorSpec` record specifies a window part in the first word and an RGB value in the other three words:

```
TYPE  ColorSpec =
      RECORD
         value:   Integer;    {part identifier}
         rgb:     RGBColor;   {RGB value}
      END;
```

The `value` field of a `colorSpec` record specifies a constant that defines which part of the window the color controls. For the window color table used by the standard window definition function, you can specify these values with these meanings:

```
CONST
wContentColor     = 0;  {content region background}
wFrameColor       = 1;  {window outline}
wTextColor        = 2;  {window title and button }
                        { text}
wHiliteColor      = 3;  {reserved}
wTitleBarColor    = 4;  {reserved}
wHiliteColorLight = 5;  {lightest stripes in }
                        { title bar and lightest }
                        { dimmed text}
wHiliteColorDark  = 6;  {darkest stripes in }
                        { title bar and }
                        { darkest dimmed }
                        { text}
wTitleBarLight    = 7;  {lightest parts of }
                        { title bar background}
wTitleBarDark     = 8;  {darkest parts of }
                        { title bar background}
wDialogLight      = 9;  {lightest element }
                        { of dialog box frame}
wDialogDark       = 10; {darkest element of }
                        { dialog box frame}
wTingeLight       = 11; {lightest window tinging}
wTingeDark        = 12; {darkest window tinging}
```

Note

The part codes in System 5 and System 6 are significantly different from the part codes described here, which apply only to System 7. ◆

The window parts can appear in any order in the table.

The `rgb` field of a `ColorSpec` record contains three words of data that specify the red, green, and blue values of the color to be used. The `RGBColor` data type is defined in *Inside Macintosh: Imaging*.

When your application creates a window, the Window Manager first looks for a resource of type 'wctb' with the same resource ID as the 'WIND' resource used for the window. If it finds one, it creates a window color table for the window from the information in that resource, and then displays the window in those colors. If it doesn't find a window color table resource with the same resource ID as your window resource, the Window Manager uses the default system window color table, read into the heap during application startup.

After creating a window, you can change the entries in a window's window color table with the SetWinColor procedure, described on page 4-114.

See "The Window Color Table Resource" on page 4-127 for a description of the window color table resource.

The Auxiliary Window Record

The auxiliary window record specifies the color table used by a window and contains reference information used by the Dialog Manager and the Window Manager.

The Window Manager creates and maintains the information in an auxiliary window record; your application seldom, if ever, needs to access an auxiliary window record.

```
TYPE  AuxWinPtr      = ^AuxWinRec;
      AuxWinHandle   = ^AuxWinPtr;

      AuxWinRec =
      RECORD
         awNext:        AuxWinHandle;  {handle to next record}
         awOwner:       WindowPtr;     {pointer to window }
                                       { associated with this }
                                       { record}
         awCTable:      CTabHandle;    {handle to color table}
         dialogCItem:   Handle;        {storage used by }
                                       { Dialog Manager}
         awFlags:       LongInt;       {reserved}
         awReserved:    CTabHandle;    {reserved}
         awRefCon:      LongInt;       {reference constant, }
                                       { for application's use}
      END;
```

Field descriptions

awNext — A handle to the next record in the auxiliary window list, used by the Window Manager to maintain the auxiliary window list as a linked list. If a window is using the default auxiliary window record, this value is NIL.

awOwner — A pointer to the window that uses this record. The awOwner field of the default auxiliary window record is set to NIL.

`awCTable`	A handle to the window's color table. Unless you specify otherwise, this is a handle to the system window color table.
`dialogCItem`	Private storage for use by the Dialog Manager.
`awFlags`	Reserved.
`awReserved`	Reserved.
`awRefCon`	The reference constant, typically used by an application to associate the auxiliary window record with a document record.

Except in unusual circumstances, your application doesn't need to manipulate window color tables or the auxiliary window record.

For compatibility with other applications in the shared environment, your application should not manipulate system color tables directly but should go through the Palette Manager, documented in *Inside Macintosh: Imaging*. If your application provides its own window and control definition functions, these functions should apply the user's desktop color choices the same way the standard window and control definition functions do.

The Window List

The Window Manager maintains information about the windows on the desktop in a private structure called the *window list*. The window list contains pointers to all windows on the desktop, both visible and invisible, and contains other information that the Window Manager uses to maintain the desktop.

Your application should not directly access the information in a window list. The structure of the window list is private to the Window Manager.

The global variable `WindowList` contains a pointer to the first window in the window list.

Window Manager Routines

This section describes the complete set of routines for creating, displaying, and managing windows.

Initializing the Window Manager

Before using any other other Window Manager routines, you must initialize the Window Manager by calling the `InitWindows` procedure.

As part of initialization, `InitWindows` creates the **Window Manager port,** a graphics port that occupies all of the main screen. The Window Manager port is named `WMgrCPort` on Macintosh computers equipped with Color QuickDraw and `WMgrPort` on computers with only QuickDraw.

Ordinarily, your application does not need to know about the Window Manager port. If necessary, however, you can retrieve a pointer to it by calling the procedure `GetWMgrPort` or `GetCWMgrPort`. Your application should not draw directly into the Window Manager port, except through custom window definition functions.

The Window Manager draws your application's windows into the Window Manager port. The port rectangle of the Window Manager port is the bounding rectangle of the main screen (`screenBits.bounds`). To accommodate systems with multiple monitors, QuickDraw recognizes a port rectangle of `screenBits.bounds` as a special case and allows drawing on all parts of the desktop.

InitWindows

The procedure `InitWindows` initializes the Window Manager for your application. Before calling `InitWindows`, you must initialize QuickDraw and the Font Manager by calling the `InitGraf` and `InitFonts` procedures, documented in *Inside Macintosh: Imaging* and *Inside Macintosh: Text*.

```
PROCEDURE InitWindows;
```

DESCRIPTION

The `InitWindows` procedure initializes the Window Manager.

ASSEMBLY-LANGUAGE INFORMATION

When the desktop needs to be redrawn any time after initialization, the Window Manager checks the global variable `DeskHook`, which can be used as a pointer to an application-defined routine for drawing the desktop. This variable is ordinarily set to 0, but not until after system startup. If you're displaying windows in code that is to be executed during startup, set `DeskHook` to 0. Note that the use of the Window Manager's global variables is not guaranteed to be compatible in system software versions later than System 6.

Creating Windows

You can create windows in two ways:

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

Creating windows from resources allows you to localize your application for different languages and to change the characteristics of your windows during application development by changing only the window resources.

All four functions, GetNewCWindow, GetNewWindow, NewCWindow, and NewWindow, can allocate space in your application's heap for the new window's window record. For more control over memory use, you can allocate the space yourself and pass a pointer when creating a window. In either case, the Window Manager fills in the data structure and returns a pointer to it.

GetNewCWindow

Use the GetNewCWindow function to create a color window with the properties defined in the 'WIND' resource with a specified resource ID.

```
FUNCTION GetNewCWindow (windowID: Integer; wStorage: Ptr;
                        behind: WindowPtr): WindowPtr;
```

windowID — The resource ID of the 'WIND' resource that defines the properties of the window.

wStorage — A pointer to memory space for the window record.

If you specify a value of NIL for wStorage, the GetNewCWindow function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the wStorage parameter.

behind — A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of Pointer(–1). When you place a window in front of all others, GetNewCWindow removes the highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of NIL.

DESCRIPTION

The GetNewCWindow function creates a new color window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If GetNewCWindow is unable to read the window or window definition function from the resource file, it returns NIL.

The GetNewCWindow function looks for a 'wctb' resource with the same resource ID as that of the 'WIND' resource. If it finds one, it uses the window color information in the 'wctb' resource for coloring the window frame and highlighting selected text.

If the window's definition function (specified in the window resource) is not already in memory, GetNewCWindow reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure and content regions of the window and asks the window definition function to calculate those regions.

To create the window, GetNewCWindow retrieves the window characteristics from the window resource and then calls the NewCWindow function, passing the characteristics as parameters.

The GetNewCWindow function creates a window in a color graphics port. Before calling GetNewCWindow, verify that Color QuickDraw is available. Your application typically sets up its own global variables reflecting the system setup during initialization by calling the Gestalt function. See *Inside Macintosh: Overview* for more information about establishing the local configuration.

SPECIAL CONSIDERATIONS

Note that the GetNewCWindow function returns a value of type WindowPtr, not CWindowPtr.

If you let the Window Manager create the window record in your application's heap, call DisposeWindow to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to the storage to GetNewCWindow, use the procedure CloseWindow to close the window and the procedure DisposePtr, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

See Listing 4-3 on page 4-28 for an example that calls GetNewCWindow to create a new window from a window resource.

For more information about window characteristics and the window resource, see the description of NewCWindow beginning on page 4-79 and the description of the 'WIND' resource in the section "The Window Resource" beginning on page 4-124.

For the procedures for closing a window and removing the structures from memory, see the descriptions of the DisposeWindow procedure on page 4-105, the CloseWindow procedure on page 4-104, and the DisposePtr procedure in *Inside Macintosh: Memory*. See Listing 4-17 on page 4-61 for an example of closing a document window.

GetNewWindow

Use the GetNewWindow function to create a new window from a window resource when Color QuickDraw is not available. The GetNewWindow function takes the same parameters as GetNewCWindow and returns a value of type WindowPtr. The only difference is that it creates a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics ports are the same size and can be used interchangeably in most Window Manager routines.

```
FUNCTION GetNewWindow (windowID: Integer; wStorage: Ptr;
                       behind: WindowPtr): WindowPtr;
```

windowID — The resource ID of the 'WIND' resource that defines the properties of the window.

wStorage — A pointer to memory space for the window record.

If you specify a value of NIL for wStorage, the GetNewWindow function allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the memory your application needs for window records early in your initialization code. Whenever you need to create a window, you can allocate memory from your own block and pass a pointer to it in the wStorage parameter.

behind — A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of Pointer(–1). When you place a window in front of all others, GetNewWindow removesthe highlighting from the previously active window, highlights the newly created window, and generates the appropriate activate events. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of NIL.

DESCRIPTION

Like GetNewCWindow, GetNewWindow creates a new window from a window resource, but it creates a monochrome window. The GetNewWindow function creates a new window from the specified window resource and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If GetNewWindow is unable to read the window or window definition function from the resource file, it returns NIL.

If the window's definition function (specified in the window resource) is not already in memory, GetNewWindow reads it into memory and stores a handle to it in the window record. It allocates space in the application heap for the structure and content regions of the window and asks the window definition function to calculate those regions.

To create the window, GetNewWindow retrieves the window characteristics from the window resource and then calls the function NewWindow, passing the characteristics as parameters.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call DisposeWindow to dispose of the window's window record. If you allocated the memory for the window record yourself and passed a pointer to GetNewWindow, use the procedure CloseWindow to close the window and the procedure DisposePtr, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For more information about window characteristics and the window resource, see the description of NewWindow beginning on page 4-82 and the description of the 'WIND' resource in the section "The Window Resource" beginning on page 4-124.

For the procedures for closing a window and removing the structures from memory, see the descriptions of the DisposeWindow procedure on page 4-105, the CloseWindow procedure on page 4-104, and the DisposePtr procedure in *Inside Macintosh: Memory*.

NewCWindow

You can use the NewCWindow function to create a window with a specified list of characteristics.

```
FUNCTION NewCWindow (wStorage: Ptr; boundsRect: Rect;
                     title: Str255; visible: Boolean;
                     procID: Integer; behind: WindowPtr;
                     goAwayFlag: Boolean;
                     refCon: LongInt): WindowPtr;
```

wStorage — A pointer to the window record. If you specify NIL as the value of wStorage, NewCWindow allocates the window record as a nonrelocatable object in the application heap. You can reduce the chances of heap fragmentation by allocating memory from a block of memory reserved for this purpose by your application and passing a pointer to it in the wStorage parameter.

boundsRect — A rectangle, in global coordinates, specifying the window's initial size and location. This parameter becomes the port rectangle of the window's graphics port. For the standard window types, the boundsRect field defines the content region of the window. The NewCWindow function places the origin of the local coordinate system at the upper-left corner of the port rectangle.

Note

The NewCWindow function actually calls the QuickDraw procedure OpenCPort to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by OpenCPort, except for the character font, which is set to the application font instead of the system font. ◆

`title` A string that specifies the window's title.

If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters are truncated at the end of the title; if there's no close box, the title is centered and truncated at both ends.

To suppress the title in a window with a title bar, pass an empty string, not NIL, in the `title` parameter.

`visible` A Boolean value indicating visibility status: TRUE means that the Window Manager displays the window; FALSE means it does not.

If the value of the `visible` parameter is TRUE, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the goAwayFlag parameter is also TRUE and the window is frontmost (that is, if the value of the behind parameter is Pointer(–1)), the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region.

When you create a window, you typically specify FALSE as the value of the `visible` parameter. When you're ready to display the window, you call the ShowWindow procedure, described on page 4-88.

`procID` The window's definition ID, which specifies both the window definition function and the variation code within that definition function.

The Window Manager supports nine standard window types, which are handled by two window definition functions. You can create windows of the standard types by specifying one of the window definition ID constants:

```
CONST
   documentProc      = 0;  {standard document }
                           { window, no zoom box}
   dBoxProc          = 1;  {alert box or modal }
                           { dialog box}
   plainDBox         = 2;  {plain box}
   altDBoxProc       = 3;  {plain box with shadow}
   noGrowDocProc     = 4;  {movable window, }
                           { no size box or zoom box}
   movableDBoxProc   = 5;  {movable modal dialog box}
   zoomDocProc       = 8;  {standard document window}
   zoomNoGrow        = 12; {zoomable, nonresizable }
                           { window}
   rDocProc          = 16; {rounded-corner window}
```

For a description of the nine standard window types, see "Types of Windows" beginning on page 4-8.

You can control the diameter of curvature of rounded-corner windows by adding an integer to the `rDocProc` constant, as described in "The Window Resource" beginning on page 4-124.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(–1)`. When you place a new window in front of all others, `NewCWindow` removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application's updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

`goAwayFlag` A Boolean value that determines whether the window has a close box. If the value of `goAwayFlag` is `TRUE` and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of `goAwayFlag` is `FALSE` or the window type does not support a close box, it does not.

`refCon` The window's reference constant, set and used only by your application. (See "Managing Multiple Windows" beginning on page 4-23 for some suggested ways to use the `refCon` parameter.)

DESCRIPTION

The `NewCWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `NewCWindow` is unable to read the window definition function from the resource file, it returns `NIL`.

The `NewCWindow` function looks for a `'wctb'` resource with the same resource ID as the `'WIND'` resource. If it finds one, it uses the window color information in the `'wctb'` resource for coloring the window frame and highlighting.

If the window's definition function is not already in memory, `NewCWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

The `NewCWindow` function creates a window in a color graphics port. Creating color windows whenever possible ensures that your windows appear on color monitors with whatever color options the user has selected. Before calling `GetNewCWindow`, verify that Color QuickDraw is available. Your application typically sets up its own set of global

variables reflecting the system setup during initialization by calling the `Gestalt` function. See the chapter *Inside Macintosh: Overview* for more information about establishing the local configuration.

Note that the function `NewCWindow` returns a value of type `WindowPtr`, not `CWindowPtr`.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call the `DisposeWindow` procedure to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewCWindow`, use the `CloseWindow` procedure to close the window and the `DisposePtr` procedure, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*.

NewWindow

Use the `NewWindow` function to create a new window with the characteristics specified by a list of parameters when Color QuickDraw is not available. The `NewWindow` function takes the same parameters as `NewCWindow` and, like `NewCWindow`, returns a `WindowPtr` as its function result. The only difference is that `NewWindow` creates a window in a monochrome graphics port, not a color graphics port. The window record and graphics port record that describe monochrome and color graphics ports are the same size and can be used interchangeably in most Window Manager routines.

```
FUNCTION NewWindow (wStorage: Ptr; boundsRect: Rect;
                    title: Str255; visible: Boolean;
                    theProc: Integer; behind: WindowPtr;
                    goAwayFlag: Boolean;
                    refCon: LongInt): WindowPtr;
```

`wStorage` A pointer to the window record. If you specify `NIL` as the value of `wStorage`, `NewWindow` allocates the window record as a nonrelocatable object in the heap. You can reduce the chances of heap fragmentation by allocating the storage from a block of memory reserved for this purpose by your application and passing a pointer to it in the `wStorage` parameter.

`boundsRect` A rectangle, in global coordinates, specifying the window's initial size and location. This parameter becomes the port rectangle of the window's graphics port. For the standard window types, `boundsRect` defines the content region of the window. The `NewWindow` function places the origin of the local coordinate system at the upper-left corner of the port rectangle.

Note

The `NewWindow` function actually calls the QuickDraw procedure `OpenPort` to create the graphics port. The bitmap, pen pattern, and other characteristics of the window's graphics port are the same as the default values set by `OpenPort`, except for the character font, which is set to the application font instead of the system font. The coordinates of the graphics port's port boundaries and visible region are changed along with its port rectangle. ◆

`title` A string that specifies the window's title.

If the title is too long to fit in the title bar, the title is truncated. If the window has a close box, characters at the end of the title are truncated; if there's no close box, the title is centered and truncated at both ends.

To suppress the title in a window with a title bar, pass an empty string, not `NIL`.

`visible` A Boolean value indicating visibility status: `TRUE` means that the Window Manager displays the window; `FALSE` means it does not.

If the value of the `visible` parameter is `TRUE`, the Window Manager draws a new window as soon as the window exists. The Window Manager first calls the window definition function to draw the window frame. If the value of the `goAwayFlag` parameter (described below) is also `TRUE` and the window is frontmost (that is, if the value of the `behind` parameter is `Pointer(-1)`), the Window Manager instructs the window definition function to draw a close box in the window frame. After drawing the frame, the Window Manager generates an update event to trigger your application's drawing of the content region.

When you create a window, you typically specify `FALSE` as the value of the `visible` parameter. When you're ready to display the window, you call the `ShowWindow` procedure, described on page 4-88.

`theProc` The window's definition ID, which specifies both the window definition function and the variation code for that definition function.

The Window Manager supports nine standard window types, which are handled by two window definition functions. You can create windows of the standard types by specifying one of the type constants:

```
CONST
   documentProc      = 0;  {standard document }
                           { window, no zoom box}
   dBoxProc          = 1;  {alert box or modal }
                           { dialog box}
   plainDBox         = 2;  {plain box}
```

```
altDBoxProc       = 3;  {plain box with shadow}
noGrowDocProc     = 4;  {movable window, }
                        { no size box or zoom box}
movableDBoxProc   = 5;  {movable modal dialog box}
zoomDocProc       = 8;  {standard document window}
zoomNoGrow        = 12; {zoomable, nonresizable }
                        { window}
rDocProc          = 16; {rounded-corner window}
```

You can control the diameter of curvature of rounded-corner windows by adding an integer to the `rDocProc` constant, as described in "The Window Resource" beginning on page 4-124.

`behind` A pointer to the window that appears immediately in front of the new window on the desktop.

To place a new window in front of all other windows on the desktop, specify a value of `Pointer(-1)`. When you place a new window in front of all others, `NewWindow` removes highlighting from the previously active window, highlights the newly created window, and generates activate events that trigger your application's updating of both windows. Note that if you create an invisible window in front of all others on the desktop, the user sees no active window until you make the new window visible (or make another window active).

To place a new window behind all other windows, specify a value of `NIL`.

`goAwayFlag` A Boolean value that determines whether or not the window has a close box. If the value of `goAwayFlag` is `TRUE` and the window type supports a close box, the Window Manager draws a close box in the title bar and recognizes mouse clicks in the close region; if the value of `goAwayFlag` is `FALSE` or the window type does not support a close box, it does not.

`refCon` The window's reference constant, set and used only by your application. (See "Managing Multiple Windows" beginning on page 4-23 for some suggested ways to use the `refCon` parameter.)

DESCRIPTION

The `NewWindow` function creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the newly created window record. You can use the returned window pointer to refer to this window in most Window Manager routines. If `NewWindow` is unable to read the window definition function from the resource file, it returns `NIL`.

If the window's definition function is not already in memory, `NewWindow` reads it into memory and stores a handle to it in the window record. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

Storing the characteristics of your windows as resources, especially window titles and window items, makes your application easier to localize.

SPECIAL CONSIDERATIONS

If you let the Window Manager create the window record in your application's heap, call the `DisposeWindow` procedure to close the window and dispose of its window record. If you allocated the memory for the window record yourself and passed a pointer to `NewCWindow`, use the `CloseWindow` procedure to close the window and the `DisposePtr` procedure, documented in *Inside Macintosh: Memory*, to dispose of the window record.

SEE ALSO

For the procedures for closing a window and removing the structures from memory, see the descriptions of the `DisposeWindow` procedure on page 4-105, the `CloseWindow` procedure on page 4-104, and the `DisposePtr` procedure in *Inside Macintosh: Memory*.

Naming Windows

This section describes the procedures that set and retrieve a window's title.

SetWTitle

Use the `SetWTitle` procedure to change a window's title.

```
PROCEDURE SetWTitle (theWindow: WindowPtr; title: Str255);
```

`theWindow` A pointer to the window's window record.

`title` The new window title.

DESCRIPTION

The `SetWTitle` procedure changes a window's title to the specified string, both in the window record and on the screen, and redraws the window's frame as necessary.

When the user opens a previously saved document, you typically create a new (invisible) window with the title "untitled" and then call `SetWTitle` to give the window the document's name before displaying it. You also call `SetWTitle` when the user saves a document under a new name.

To suppress the title in a window with a title bar, pass an empty string, not `NIL`.

Always use `SetWTitle` instead of directly changing the title in a window's window record.

GetWTitle

Use the `GetWTitle` procedure to retrieve a window's title.

```
PROCEDURE GetWTitle (theWindow: WindowPtr; VAR title: Str255);
```

`theWindow` A pointer to the window record.
`title` The window title.

DESCRIPTION

The `GetWTitle` procedure returns the title of the window in the `title` parameter.

Your application seldom needs to determine a window's title. It might need to do so, however, when presenting user dialog boxes during operations that can affect multiple files. A spell-checking command, for example, might display a dialog box that lets the user select from all currently open documents.

When you need to retrieve a window's title, you should always use `GetWTitle` instead of reading the title from a window's window record.

Displaying Windows

This section describes the Window Manager routines that change a window's display and position in the window list but not its size or location on the desktop. Note that the Window Manager automatically draws all visible windows on the screen.

Your application typically uses only a few of the routines described in this section: `DrawGrowIcon`, `SelectWindow`, `ShowWindow`, and, occasionally, `HideWindow`.

DrawGrowIcon

Use the `DrawGrowIcon` procedure to draw a window's size box.

```
PROCEDURE DrawGrowIcon (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record.

DESCRIPTION

The `DrawGrowIcon` procedure draws a window's size box or, if the window can't be sized, whatever other image is appropriate. You call `DrawGrowIcon` when drawing the content region of a window that contains a size box.

The exact appearance and location of the image depend on the window type and the window's active or inactive state. The `DrawGrowIcon` procedure automatically checks the window's type and state and draws the appropriate image.

In an active document window, `DrawGrowIcon` draws the grow image in the size box in the lower-right corner of the window's graphics port rectangle, along with the lines delimiting the size box and scroll bar areas. To draw the size box but not the scroll bar outline, set the `clipRgn` field in the window's graphics port to be a 15-by-15 pixel rectangle in the lower-right corner of the window.

The `DrawGrowIcon` procedure doesn't erase the scroll bar areas. If you use `DrawGrowIcon` to draw the size box and scroll bar outline, therefore, you should erase those areas yourself when the window size changes, even if the window doesn't contain scroll bars.

In an inactive document window, `DrawGrowIcon` draws the lines delimiting the size box and scroll bar areas and erases the size box.

SEE ALSO

See Listing 4-8 on page 4-39 for an example that draws a window's content region, including the size box. See Listing 4-11 on page 4-51 for an example that calls `DrawGrowIcon` to remove the size-box icon when a window becomes inactive.

SelectWindow

Use the `SelectWindow` procedure to make a window active. The `SelectWindow` procedure changes the active status of a window but does not affect its visibility.

```
PROCEDURE SelectWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `SelectWindow` procedure removes highlighting from the previously active window, brings the specified window to the front, highlights it, and generates the activate events to deactivate the previously active window and activate the specified window. If the specified window is already active, `SelectWindow` has no effect.

Even if the specified window is invisible, `SelectWindow` brings the window to the front, activates the window, and deactivates the previously active window. Note that in this case, no active window is visible on the screen. If you do select an invisible window, be sure to call `ShowWindow` immediately to make the window visible (and accessible to the user).

Call `SelectWindow` when the user presses the mouse button while the cursor is in the content region of an inactive window.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `SelectWindow` to change the active window when the user presses the mouse button while the cursor is in an inactive window.

See Listing 4-18 on page 4-64 for an example that uses `SelectWindow` and `ShowWindow` together to restore a window's active, visible status after it has been made invisible with `HideWindow`.

ShowWindow

Use the `ShowWindow` procedure to make an invisible window visible.

```
PROCEDURE ShowWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record of the window.

DESCRIPTION

The `ShowWindow` procedure makes an invisible window visible. If the specified window is already visible, `ShowWindow` has no effect. Your application typically creates a new window in an invisible state, performs any necessary setup of the content region, and then calls `ShowWindow` to make the window visible.

When you display a previously invisible window by calling `ShowWindow`, the Window Manager draws the window frame and then generates an update event to trigger your application's drawing of the content region.

If the newly visible window is the frontmost window, `ShowWindow` highlights it if it's not already highlighted and generates an activate event to make it active. The `ShowWindow` procedure does not activate a window that is not frontmost on the desktop.

Note
Because `ShowWindow` does not change the front-to-back ordering of windows, it is not the inverse of `HideWindow`. If you make the frontmost window invisible with `HideWindow`, and `HideWindow` has activated another window, you must call both `ShowWindow` and `SelectWindow` to bring the original window back to the front. ◆

SEE ALSO

See Listing 4-16 on page 4-60 for an example that temporarily hides a dialog box window when the user closes it. See Listing 4-18 on page 4-64 for the example that calls `ShowWindow` to display the window again later.

HideWindow

Use the HideWindow procedure to make a window invisible.

```
PROCEDURE HideWindow (theWindow: WindowPtr);
```

theWindow A pointer to the window's window record.

DESCRIPTION

The HideWindow procedure make a visible window invisible. If you hide the frontmost window, HideWindow removes the highlighting, brings the window behind it to the front, highlights the new frontmost window, and generates the appropriate activate events.

To reverse the actions of HideWindow, you must call both ShowWindow, to make the window visible, and SelectWindow, to select it.

SEE ALSO

See Listing 4-16 on page 4-60 for an example that calls HideWindow to temporarily hide a dialog box window when the user closes it. See Listing 4-18 on page 4-64 for the companion example that redisplays the window later.

ShowHide

Use the ShowHide procedure to set a window's visibility status.

```
PROCEDURE ShowHide (theWindow: WindowPtr; showFlag: Boolean);
```

theWindow A pointer to the window's window record.

showFlag A Boolean value that determines visibility status: TRUE makes a window visible; FALSE makes it invisible.

DESCRIPTION

The ShowHide procedure sets a window's visibility to the status specified by the showFlag parameter. If the value of showFlag is TRUE, ShowHide makes the window visible if it's not already visible and has no effect if it's already visible. If the value of showFlag is FALSE, ShowHide makes the window invisible if it's not already invisible and has no effect if it's already invisible.

The ShowHide procedure never changes the highlighting or front-to-back ordering of windows and generates no activate events.

▲ **WARNING**
Use this procedure carefully and only in special circumstances where you need more control than that provided by `HideWindow` and `ShowWindow`. Do not, for example, use `ShowHide` to hide the active window without making another window active. ▲

HiliteWindow

Use the `HiliteWindow` procedure to set a window's highlighting status.

```
PROCEDURE HiliteWindow (theWindow: WindowPtr; fHilite: Boolean);
```

`theWindow` A pointer to the window's window record.

`fHilite` A Boolean value that determines the highlighting status: `TRUE` highlights a window; `FALSE` removes highlighting.

DESCRIPTION

The `HiliteWindow` procedure sets a window's highlighting status to the specified state. If the value of the `fHilite` parameter is `TRUE`, `HiliteWindow` highlights the specified window; if the specified window is already highlighted, the procedure has no effect. If the value of `fHilite` is `FALSE`, `HiliteWindow` removes highlighting from the specified window; if the window is not already highlighted, the procedure has no effect.

Your application doesn't normally need to call `HiliteWindow`. To make a window active, you can call `SelectWindow`, which handles highlighting for you.

BringToFront

Use the `BringToFront` procedure to bring a window to the front.

```
PROCEDURE BringToFront (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `BringToFront` procedure puts the specified window at the beginning of the window list and redraws the window in front of all others on the screen. It does not change the window's highlighting or make it active.

Your application does not ordinarily call `BringToFront`. The user interface guidelines specify that the frontmost window should be the active window. To bring a window to the front and make it active, call the `SelectWindow` procedure.

SendBehind

Use the SendBehind procedure to move one window behind another.

```
PROCEDURE SendBehind (theWindow, behindWindow: WindowPtr);
```

`theWindow` A pointer to the window to be moved.

`behindWindow`
A pointer to the window that is to be in front of the moved window.

DESCRIPTION

The SendBehind procedure moves the window pointed to by the parameter `theWindow` behind the window pointed to by the parameter `behindWindow`. If the move exposes previously obscured windows or parts of windows, SendBehind redraws the frames as necessary and generates the appropriate update events to have any newly exposed content areas redrawn.

If the value of `behindWindow` is `NIL`, SendBehind sends the window to be moved behind all other windows on the desktop. If the window to be moved is the active window, SendBehind removes its highlighting, highlights the newly exposed frontmost window, and generates the appropriate activate events.

Note
Do not use SendBehind to deactivate a window after you've made a new window active with the `SelectWindow` procedure. The `SelectWindow` procedure automatically deactivates the previously active window. ◆

Retrieving Window Information

This section describes

- the `FindWindow` function, which maps the cursor location of a mouse-down event to parts of the screen or regions of a window
- the `FrontWindow` function, which tells your application which window is active

FindWindow

When your application receives a mouse-down event, call the `FindWindow` function to map the location of the cursor to a part of the screen or a region of a window.

```
FUNCTION FindWindow (thePoint: Point;
                     VAR theWindow: WindowPtr): Integer;
```

thePoint The point, in global coordinates, where the mouse-down event occurred. Your application retrieves this information from the `where` field of the event record.

theWindow A parameter in which `FindWindow` returns a pointer to the window in which the mouse-down event occurred, if it occurred in a window. If it didn't occur in a window, `FindWindow` sets `theWindow` to `NIL`.

DESCRIPTION

The `FindWindow` function returns an integer that specifies where the cursor was when the user pressed the mouse button. You typically call `FindWindow` whenever you receive a mouse-down event. The `FindWindow` function helps you dispatch the event by reporting whether the cursor was in the menu bar or in a window when the mouse button was pressed and, if it was in a window, which window and which region of the window. If the mouse-down event occurred in a window, `FindWindow` places a pointer to the window in the parameter `theWindow`.

The `FindWindow` function returns an integer that specifies one of nine regions:

```
CONST inDesk      = 0;  {none of the following}
      inMenuBar   = 1;  {in menu bar}
      inSysWindow = 2;  {in desk accessory window}
      inContent   = 3;  {anywhere in content region except size }
                        { box if window is active, }
                        { anywhere including size box if window }
                        { is inactive}
      inDrag      = 4;  {in drag (title bar) region}
      inGrow      = 5;  {in size box (active window only)}
      inGoAway    = 6;  {in close box}
      inZoomIn    = 7;  {in zoom box (window in standard state)}
      inZoomOut   = 8;  {in zoom box (window in user state)}
```

The `FindWindow` function returns `inDesk` if the cursor is not in the menu bar, a desk accessory window, or any window that belongs to your application. The `FindWindow` function might return this value if, for example, the user presses the mouse button while the cursor is on the window frame but not in the title bar, close box, or zoom box. When `FindWindow` returns `inDesk`, your application doesn't need to do anything. In System 7, when the user presses the mouse button while the cursor is on the desktop or in a window that belongs to another application, the Event Manager sends your application a suspend event and switches to the Finder or another application.

The `FindWindow` function returns `inMenuBar` when the user presses the mouse button with the cursor in the menu bar. Your application typically adjusts its menus and then calls the Menu Manager's function `MenuSelect` to let the user choose menu items.

The FindWindow function returns `inSysWindow` when the user presses the mouse button while the cursor is in a window belonging to a desk accessory that was launched in your application's partition. This situation seldom arises in System 7. When the user

clicks in a window belonging to a desk accessory launched independently, the Event Manager sends your application a suspend event and switches to the desk accessory.

If `FindWindow` does return `inSysWindow`, your application calls the `SystemClick` procedure, documented in the chapter "Event Manager" in this book. The `SystemClick` procedure routes the event to the desk accessory. If the user presses the mouse button with the cursor in the content region of an inactive desk accessory window, `SystemClick` makes the window active by sending your application and the desk accessory the appropriate activate events.

The `FindWindow` function returns `inContent` when the user presses the mouse button with the cursor in the content area (excluding the size box in an active window) of one of your application's windows. Your application then calls its routine for handling clicks in the content region.

The `FindWindow` function returns `inDrag` when the user presses the mouse button with the cursor in the drag region of a window (that is, the title bar, excluding the close box and zoom box). Your application then calls the Window Manager's `DragWindow` procedure to let the user drag the window to a new location.

The `FindWindow` function returns `inGrow` when the user presses the mouse button with the cursor in an active window's size box. Your application then calls its own routine for resizing a window.

The `FindWindow` function returns `inGoAway` when the user presses the mouse button with the cursor in an active window's close box. Your application calls the `TrackGoAway` function to track mouse activity while the button is down and then calls its own routine for closing a window if the user releases the button while the cursor is in the close box.

The `FindWindow` function returns `inZoomIn` or `inZoomOut` when the user presses the mouse button with the cursor in an active window's zoom box. Your application calls the `TrackBox` function to track mouse activity while the button is down and then calls its own routine for zooming a window if the user releases the button while the cursor is in the zoom box.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `FindWindow` to determine the location of the cursor and then dispatches the mouse-down event depending on the results.

FrontWindow

Use the `FrontWindow` function to find out which window is active.

```
FUNCTION FrontWindow: WindowPtr;
```

DESCRIPTION

The `FrontWindow` function returns a pointer to the first visible window in the window list (that is, the active window). If there are no visible windows, `FrontWindow` returns `NIL`.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `FrontWindow` to determine whether an event occurred in the active window.

See Listing 4-12 on page 4-55 for an example that calls `FrontWindow` to determine whether to display a window in front of other windows after changing its size.

See Listing 4-16 on page 4-60 and Listing 4-17 on page 4-61 for examples that call `FrontWindow` to determine which window is affected by a user command directed at the active window.

Moving Windows

This section describes the procedures that move windows on the desktop.

To move a window, your application ordinarily needs to call only the `DragWindow` procedure, which itself calls the `DragGrayRgn` function, and the `MoveWindow` procedure. The `DragGrayRgn` function drags a dotted outline of the window on the screen, following the motion of the cursor, as long as the user holds down the mouse button. The `DragGrayRgn` function itself calls the `PinRect` function to contain the point where the cursor was when the mouse button was first pressed inside the available desktop area. When the user releases the mouse button, `DragWindow` calls `MoveWindow`, which moves the window to a new location.

DragWindow

When the user drags a window by its title bar, use the `DragWindow` procedure to move the window on the screen.

```
PROCEDURE DragWindow (theWindow: WindowPtr;
                      startPt: Point; boundsRect: Rect);
```

`theWindow` A pointer to the window record of the window to be dragged.

`startPt` The location, in global coordinates, of the cursor at the time the user pressed the mouse button. Your application retrieves this point from the `where` field of the event record.

`boundsRect` A rectangle, in global coordinates, that limits the region to which a window can be dragged. If the mouse button is released when the cursor is outside the limits of `boundsRect`, `DragWindow` returns without moving the window (or, if it was inactive, without making it the active window).

Because the user cannot ordinarily move the cursor off the desktop, you can safely set `boundsRect` to the largest available rectangle (the bounding box of the desktop region pointed to by the global variable `GrayRgn`) when you're using `DragWindow` to track mouse movements. Don't set the bounding rectangle to the size of the immediate screen (`screenBits.bounds`), because the user wouldn't be able to move the window to a different screen on a system equipped with multiple monitors.

DESCRIPTION

The `DragWindow` procedure moves a dotted outline of the specified window around the screen, following the movement of the cursor until the user releases the mouse button. When the button is released, `DragWindow` calls `MoveWindow` to move the window to its new location. If the specified window isn't the active window (and the Command key wasn't down when the mouse button was pressed), `DragWindow` makes it the active window by setting the `front` parameter to `TRUE` when calling `MoveWindow`. If the Command key was down when the mouse button was pressed, `DragWindow` moves the window without making it active.

SEE ALSO

The `DragWindow` procedure calls both `MoveWindow` and `DragGrayRgn`, which are described in this section.

See Listing 4-9 on page 4-44 for an example that calls `DragWindow` when the user presses the mouse button while the cursor is in the drag region.

MoveWindow

Use the `MoveWindow` procedure to move a window on the desktop.

```
PROCEDURE MoveWindow (theWindow: WindowPtr;
                      hGlobal, vGlobal: Integer;
                      front: Boolean);
```

`theWindow` A pointer to the window record of the window being moved.

`hGlobal` The new location, in global coordinates, of the left edge of the window's port rectangle.

`vGlobal` The new location, in global coordinates, of the top edge of the window's port rectangle.

front A Boolean value specifying whether the window is to become the frontmost, active window. If the value of the front parameter is FALSE, MoveWindow does not change its plane or status. If the value of the front parameter is TRUE and the window isn't active, MoveWindow makes it active by calling the SelectWindow procedure.

DESCRIPTION

The MoveWindow procedure moves the specified window to the location specified by the hGlobal and vGlobal parameters, without changing the window's size. The upper-left corner of the window's port rectangle is placed at the point (vGlobal,hGlobal). The local coordinates of the upper-left corner are unaffected.

Your application doesn't normally call MoveWindow. When the user drags a window by dragging its title bar, you can call DragWindow, which in turn calls MoveWindow when the user releases the mouse button.

DragGrayRgn

The DragWindow function calls the DragGrayRgn function to move an outline of a window around the screen as the user drags a window.

```
FUNCTION DragGrayRgn (theRgn: RgnHandle; startPt: Point;
                      limitRect, slopRect: Rect; axis: Integer;
                      actionProc: ProcPtr): LongInt;
```

theRgn A handle to the region to be dragged.

startPt The location, in the local coordinates of the current graphics port, of the cursor when the mouse button was pressed.

limitRect A rectangle, in the local coordinates of the current graphics port, that limits where the region can be dragged. This parameter works in conjunction with the slopRect parameter, as illustrated in Figure 4-23 on page 4-98.

slopRect A rectangle, in the local coordinates of the current graphics port, that gives the user some leeway in moving the mouse without violating the limits of the limitRect parameter, as illustrated in Figure 4-23 on page 4-98. The slopRect rectangle should be larger than the limitRect rectangle.

axis A constant that constrains the region's motion. The axis parameter can have one of these values:

```
CONST noConstraint   = 0;  {no constraints}
      hAxisOnly      = 1;  {move on horizontal axis }
                           { only}
      vAxisOnly      = 2;  {move on vertical axis }
                           { only}
```

If an axis constraint is in effect, the outline follows the cursor's movements along only the specified axis, ignoring motion along the other axis. With or without an axis constraint, the outline appears only when the mouse is inside the `slopRect` rectangle.

`actionProc` A pointer to a procedure that defines an action to be performed repeatedly as long as the user holds down the mouse button. The procedure can have no parameters. If the value of `actionProc` is `NIL`, `DragGrayRgn` simply retains control until the mouse button is released.

DESCRIPTION

The `DragGrayRgn` function moves a gray outline of a region on the screen, following the movements of the cursor, until the mouse button is released. It returns the difference between the point where the mouse button was pressed and the **offset point**—that is, the point in the region whose horizontal and vertical offsets from the upper-left corner of the region's enclosing rectangle are the same as the offsets of the starting point when the user pressed the mouse button. The `DragGrayRgn` function stores the vertical difference between the starting point and the offset point in the high-order word of the return value and the horizontal difference in the low-order word.

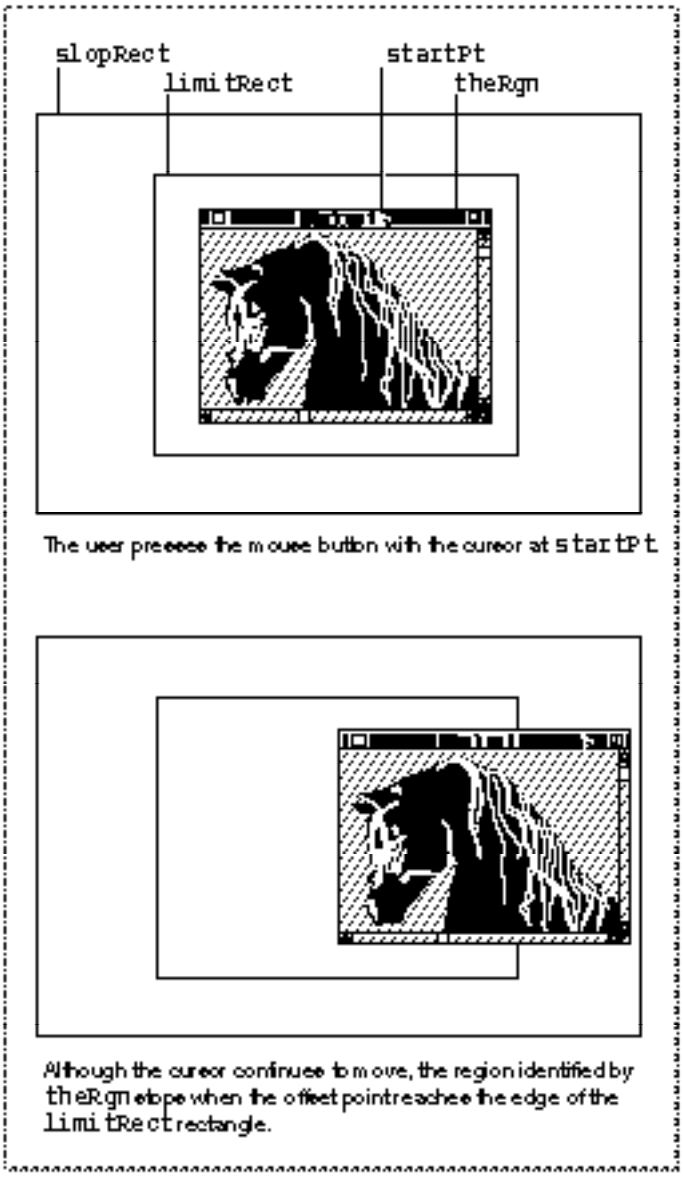
The `DragGrayRgn` function limits the movement of the region according to the constraints set by the `limitRect` and `slopRect` parameters:

- As long as the cursor is inside the `limitRect` rectangle, the region's outline follows it normally. If the mouse button is released while the cursor is within this rectangle, the return value reflects the simple distance that the cursor moved in each dimension.
- When the cursor moves outside the `limitRect` rectangle, the offset point stops at the edge of the `limitRect` rectangle. If the mouse button is released while the cursor is outside the `limitRect` rectangle but inside the `slopRect` rectangle, the return value reflects only the difference between the starting point and the offset point, regardless of how far outside of the `limitRect` rectangle the cursor may have moved. (Note that part of the region can fall outside the `limitRect` rectangle, but not the offset point.)
- When the cursor moves outside the `slopRect` rectangle, the region's outline disappears from the screen. The `DragGrayRgn` function continues to track the cursor, however, and if the cursor moves back into the `slopRect` rectangle, the outline reappears. If the mouse button is released while the cursor is outside the `slopRect` rectangle, both words of the return value are set to $8000. In this case, the Window Manager does not move the window from its original location.

Figure 4-23 on page 4-98 illustrates how the region stops moving when the offset point reaches the edge of the `limitRect` rectangle. The cursor continues to move, but the region does not.

If the mouse button is released while the cursor is anywhere inside the `slopRect` rectangle, the Window Manager redraws the window in its new location, which is calculated from the value returned by `DragGrayRgn`.

Figure 4-23 Limiting rectangle used by `DragGrayRgn`



ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `DragGrayRgn` as long as the mouse button is held down. (If there's an `actionProc` procedure, it is called first.) If you want `DragGrayRgn` to draw the region's outline in a pattern other than gray, you can store the pattern in the global variable `DragPattern` and then invoke the macro `_DragTheRgn`. Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

PinRect

The `DragGrayRgn` function uses the `PinRect` function to contain a point within a specified rectangle.

```
FUNCTION PinRect (theRect: Rect; thePt: Point): LongInt;
```

`theRect`	The rectangle in which the point is to be contained.
`thePt`	The point to be contained.

DESCRIPTION

The `PinRect` function returns a point within the specified rectangle that is as close as possible to the specified point. (The high-order word of the returned long integer is the vertical coordinate; the low-order word is the horizontal coordinate.)

If the specified point is within the rectangle, `PinRect` returns the point itself. If not, then

- if the horizontal position is to the left of the rectangle, `PinRect` returns the left edge as the horizontal coordinate
- if the horizontal position is to the right of the rectangle, `PinRect` returns the right edge minus 1 as the horizontal coordinate
- if the vertical position is above the rectangle, `PinRect` returns the top edge as the vertical coordinate
- if the vertical position is below the rectangle, `PinRect` returns the bottom edge minus 1 as the vertical coordinate

Note
The 1 is subtracted when the point is below or to the right of the rectangle so that a pixel drawn at that point lies within the rectangle. If the point is exactly on the bottom or the right edge of the rectangle, however, 1 should be subtracted but isn't. ◆

Resizing Windows

This section describes the procedures you can use to track the cursor while the user resizes a window and to draw the window in a new size.

GrowWindow

Use the `GrowWindow` function to allow the user to change the size of a window. The `GrowWindow` function displays an outline (grow image) of the window as the user moves the cursor to make the window larger or smaller; it handles all user interaction

until the user releases the mouse button. After calling `GrowWindow`, you call the `SizeWindow` procedure to change the size of the window.

```
FUNCTION GrowWindow (theWindow: WindowPtr;
                     startPt: Point; sizeRect: Rect): LongInt;
```

`theWindow` A pointer to the window record of the window to drag.

`startPt` The location of the cursor at the time the mouse button was first pressed, in global coordinates. Your application retrieves this point from the `where` field of the event record.

`sizeRect` The limits on the vertical and horizontal measurements of the port rectangle, in pixels.

Although the `sizeRect` parameter is in the form of the `Rect` data type, the four numbers in the structure represent lengths, not screen coordinates. The `top`, `left`, `bottom`, and `right` fields of the `sizeRect` parameter specify the minimum vertical measurement (`top`), the minimum horizontal measurement (`left`), the maximum vertical measurement (`bottom`), and the maximum horizontal measurement (`right`).

The minimum measurements must be large enough to allow a manageable rectangle; 64 pixels on a side is typical. Because the user cannot ordinarily move the cursor off the screen, you can safely set the upper bounds to the largest possible length (65,535 pixels) when you're using `GrowWindow` to follow cursor movements.

DESCRIPTION

The `GrowWindow` function moves a dotted-line image of the window's right and lower edges around the screen, following the movements of the cursor until the mouse button is released. It returns the new dimensions, in pixels, of the resulting window: the height in the high-order word of the returned long-integer value and the width in the low-order word. You can use the functions `HiWord` and `LoWord` to retrieve only the high-order and low-order words, respectively.

A return value of 0 means that the new size is the same as the size of the current port rectangle.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `GrowWindow` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that calls `GrowWindow` when the user presses the mouse button while the cursor is in the size box.

SizeWindow

Use the `SizeWindow` procedure to set the size of a window.

```
PROCEDURE SizeWindow (theWindow: WindowPtr; w, h: Integer;
                      fUpdate: Boolean);
```

`theWindow`	A pointer to the window record of the window to be sized.
w	The new window width, in pixels.
h	The new window height, in pixels.
`fUpdate`	A Boolean value that specifies whether any newly created area of the content region is to be accumulated into the update region (`TRUE`) or not (`FALSE`). You ordinarily pass a value of `TRUE` to ensure that the area is updated. If you pass `FALSE`, you're responsible for maintaining the update region yourself. For more information on adding rectangles to and removing rectangles from the update region, see the description of `InvalRect` on page 4-107 and `ValidRect` on page 4-108.

DESCRIPTION

The `SizeWindow` procedure changes the size of the window's graphics port rectangle to the dimensions specified by the `w` and `h` parameters, or does nothing if the values of `w` and `h` are 0. The Window Manager redraws the window in the new size, recentering the title and truncating it if necessary. Your application calls `SizeWindow` immediately after calling `GrowWindow`, to adjust the window to any changes made by the user through the size box.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that calls `SizeWindow` to resize a window based on the return value of `GrowWindow`.

Zooming Windows

This section describes the procedures you can use to track mouse activity in the zoom box and to zoom windows.

TrackBox

Use the `TrackBox` function to track the cursor when the user presses the mouse button while the cursor is in the zoom box.

```
FUNCTION TrackBox (theWindow: WindowPtr; thePt: Point;
                   partCode: Integer): Boolean;
```

theWindow A pointer to the window record of the window in which the mouse button was pressed.

thePt The location of the cursor when the mouse button was pressed. Your application receives this point from the `where` field in the event record.

partCode The part code (either `inZoomIn` or `inZoomOut`) returned by the `FindWindow` function.

DESCRIPTION

The `TrackBox` function tracks the cursor when the user presses the mouse button while the cursor is in the zoom box, retaining control until the mouse button is released. While the button is down, `TrackBox` highlights the zoom box while the cursor is in the zoom region, as illustrated in Figure 4-20 on page 4-47.

When the mouse button is released, `TrackBox` removes the highlighting from the zoom box and returns `TRUE` if the cursor is within the zoom region and `FALSE` if it is not.

Your application calls the `TrackBox` function when it receives a result code of either `inZoomIn` or `inZoomOut` from the `FindWindow` function. If `TrackBox` returns `TRUE`, your application calculates the standard state, if necessary, and calls the `ZoomWindow` procedure to zoom the window. If `TrackBox` returns `FALSE`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `TrackBox` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-12 on page 4-55 for an example that calls `TrackBox` to track cursor activity when the user presses the mouse button while the cursor is in the zoom box.

ZoomWindow

Use the `ZoomWindow` procedure to zoom the window when the user has pressed and released the mouse button with the cursor in the zoom box.

```
PROCEDURE ZoomWindow (theWindow: WindowPtr;
                      partCode: Integer; front: Boolean);
```

theWindow A pointer to the window record of the window to be zoomed.

partCode The result (either `inZoomIn` or `inZoomOut`) returned by the `FindWindow` function.

`front` A Boolean value that determines whether the window is to be brought to the front. If the value of `front` is `TRUE`, the window necessarily becomes the frontmost, active window. If the value of `front` is `FALSE`, the window's position in the window list does not change. Note that if a window was active before it was zoomed, it remains active even if the value of `front` is `FALSE`.

DESCRIPTION

The `ZoomWindow` procedure zooms a window in or out, depending on the value of the `partCode` parameter. Your application calls `ZoomWindow`, passing it the part code returned by `FindWindow`, when it receives a result of `TRUE` from `TrackBox`. The `ZoomWindow` procedure then changes the window's port rectangle to either the user state (if the part code is `inZoomIn`) or the standard state (if the part code is `inZoomOut`), as stored in the window state data record, described in the section "Zooming a Window" beginning on page 4-53.

If the part code is `inZoomOut`, your application ordinarily calculates and sets the standard state before calling `ZoomWindow`.

For best results, call the QuickDraw procedure `EraseRect`, passing the window's graphics port as the port rectangle, before calling `ZoomWindow`.

SEE ALSO

See Listing 4-12 on page 4-55 for an example that calculates and sets the standard state and then calls `ZoomWindow` to zoom a window.

Closing and Deallocating Windows

This section describes the procedures that track user activity in the close box and that close and dispose of windows.

When you no longer need a window, call the `CloseWindow` procedure if you allocated the memory for the window record or the `DisposeWindow` procedure if you did not.

TrackGoAway

Use the `TrackGoAway` function to track the cursor when the user presses the mouse button while the cursor is in the close box.

```
FUNCTION TrackGoAway (theWindow: WindowPtr;
                      thePt: Point): Boolean;
```

`theWindow` A pointer to the window record of the window in which the mouse-down event occurred.

`thePt` The location of the cursor at the time the mouse button was pressed. Your application receives this point from the `where` field of the event record.

DESCRIPTION

The `TrackGoAway` function tracks cursor activity when the user presses the mouse button while the cursor is in the close box, retaining control until the user releases the mouse button. While the button is down, `TrackGoAway` highlights the close box as long as the cursor is in the close region, as illustrated in Figure 4-19 on page 4-46.

When the mouse button is released, `TrackGoAway` removes the highlighting from the close box and returns `TRUE` if the cursor is within the close region and `FALSE` if it is not.

Your application calls the `TrackGoAway` function when it receives a result code of `inGoAway` from the `FindWindow` function. If `TrackGoAway` returns `TRUE`, your application calls its own procedure for closing a window, which can call either the `CloseWindow` procedure or the `DisposeWindow` procedure to remove the window from the screen. (Before removing a document window, your application ordinarily checks whether the document has changed since the associated file was last saved. See the chapter "Introduction to File Management" in *Inside Macintosh: Files* for a general discusion of handling files.) If `TrackGoAway` returns `FALSE`, your application does nothing.

ASSEMBLY-LANGUAGE INFORMATION

You can set the global variable `DragHook` to point to an optional procedure, defined by your application, which will be called by `TrackGoAway` as long as the mouse button is held down. (If there's an `actionProc` procedure, the `actionProc` procedure is called first.) Note that the use of the Window Manager's global variables is not guaranteed to be compatible with system software versions later than System 6.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that calls `TrackGoAway` to track cursor activity when the user presses the mouse button while the cursor is in the close box.

CloseWindow

Use the `CloseWindow` procedure to remove a window if you allocated memory yourself for the window's window record.

```
PROCEDURE CloseWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record of the window to be closed.

DESCRIPTION

The `CloseWindow` procedure removes the specified window from the screen and deletes it from the window list. It releases the memory occupied by all data structures associated with the window except the window record itself.

If you allocated memory for the window record and passed a pointer to it as one of the parameters to the functions that create windows, call `CloseWindow` when you're done with the window. You must then call the Memory Manager procedure `DisposePtr` to release the memory occupied by the window record.

▲ **WARNING**

If your application allocated any other memory for use with a window, you must release it before calling `CloseWindow`. The Window Manager releases only the data structures it created.

Also, `CloseWindow` assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the QuickDraw procedure `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` procedure and set the `windowPic` field to `NIL` before closing the window. ▲

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

SEE ALSO

See Listing 4-17 on page 4-61 for an example that calls `CloseWindow` to remove a window from the screen.

See Listing 4-3 on page 4-28 for an example that calls `CloseWindow` to clean up memory when an attempt to create a new window fails.

DisposeWindow

Use the `DisposeWindow` procedure to remove a window if you let the Window Manager allocate memory for the window record.

```
PROCEDURE DisposeWindow (theWindow: WindowPtr);
```

`theWindow` A pointer to the window record of the window to be closed.

DESCRIPTION

The `DisposeWindow` procedure removes a window from the screen, deletes it from the window list, and releases the memory occupied by all structures associated with the window, including the window record. (`DisposeWindow` calls `CloseWindow` and then releases the memory occupied by the window record.)

▲ **WARNING**
If your application allocated any other memory for use with a window, you must release it before calling `DisposeWindow`. The Window Manager releases only the data structures it created.

The `DisposeWindow` procedure assumes that any picture pointed to by the window record field `windowPic` is data, not a resource, and it calls the QuickDraw procedure `KillPicture` to delete it. If your application uses a picture stored as a resource, you must release the memory it occupies with the `ReleaseResource` procedure and set the `windowPic` field to `NIL` before closing the window. ▲

Any pending update events for the window are discarded. If the window being removed is the frontmost window, the window behind it, if any, becomes the active window.

Maintaining the Update Region

This section describes the routines you use to update your windows and to maintain window update regions.

BeginUpdate

Use the `BeginUpdate` procedure to start updating a window when you receive an update event for that window.

```
PROCEDURE BeginUpdate (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record. Your application gets this information from the `message` field in the update event record.

DESCRIPTION

The `BeginUpdate` procedure limits the visible region of the window's graphics port to the intersection of the visible region and the update region; it then sets the window's update region to an empty region. After calling `BeginUpdate`, your application redraws either the entire content region or only the visible region. In either case, only the parts of the window that require updating are actually redrawn on the screen.

Every call to `BeginUpdate` must be matched with a subsequent call to `EndUpdate` after your application redraws the content region.

Note
In Pascal, `BeginUpdate` and `EndUpdate` can't be nested. That is, you must call `EndUpdate` before the next call to `BeginUpdate`.

You can nest `BeginUpdate` and `EndUpdate` calls in assembly language if you save and restore the copy of the `visRgn`, a copy of which is stored, in global coordinates, in the global variable `SaveVisRgn`. ◆

SPECIAL CONSIDERATIONS

If you don't clear the update region when you receive an update event, the Event Manager continues to send update events until you do.

SEE ALSO

See Figure 4-21 on page 4-49 for an illustration of how `BeginUpdate` and `EndUpdate` affect the visible region and update region. See Listing 4-10 on page 4-50 for an example that updates a window.

EndUpdate

Use the `EndUpdate` procedure to finish updating a window.

```
PROCEDURE EndUpdate (theWindow: WindowPtr);
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `EndUpdate` procedure restores the normal visible region of a window's graphics port. When you receive an update event for a window, you call `BeginUpdate`, redraw the update region, and then call `EndUpdate`. Each call to `BeginUpdate` must be balanced by a subsequent call to `EndUpdate`.

SEE ALSO

See Figure 4-21 on page 4-49 for an illustration of how `BeginUpdate` and `EndUpdate` affect the visible region and update region. See Listing 4-10 on page 4-50 for an example that updates a window.

InvalRect

Use the `InvalRect` procedure to add a rectangle to a window's update region.

```
PROCEDURE InvalRect (badRect: Rect);
```

`badRect` A rectangle, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalRect` procedure adds a specified rectangle to the update region of the window whose graphics port is the current port. Specify the rectangle in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Both your application and the Window Manager use the `InvalRect` procedure. When the user enlarges a window, for example, the Window Manager uses `InvalRect` to add the newly created content region to the update region. Your application uses `InvalRect` to add the two rectangles formerly occupied by the scroll bars in the smaller content area.

InvalRgn

Use the `InvalRgn` procedure to add a region to a window's update region.

```
PROCEDURE InvalRgn (badRgn: RgnHandle);
```

`badRgn` The region, in local coordinates, that is to be added to a window's update region.

DESCRIPTION

The `InvalRgn` procedure adds a specified region to the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that uses `InvalRgn` to add part of the window's content region to the update region.

ValidRect

Use the `ValidRect` procedure to remove a rectangle from a window's update region.

```
PROCEDURE ValidRect (goodRect: Rect);
```

`goodRect` A rectangle, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRect` procedure removes a specified rectangle from the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Your application uses `ValidRect` to tell the Window Manager that it has already drawn a rectangle and to cancel any updates accumulated for that area. You can thereby improve response time by reducing redundant redrawing.

Suppose, for example, that you've resized a window that contains a size box and scroll bars. Depending on the dimensions of the newly sized window, the new size box and scroll bar areas may or may not have been accumulated into the window's update region. After calling `SizeWindow`, you can redraw the size box or scroll bars immediately and then call `ValidRect` for the areas they occupy. If they were in fact accumulated into the update region, `ValidRect` removes them so that you do not have to redraw them with the next update event.

SEE ALSO

See Listing 4-13 on page 4-58 for an example that uses `ValidRect` to remove part of the window's content region from the update region.

ValidRgn

Use the `ValidRgn` procedure to remove a specified region from a window's update region.

```
PROCEDURE ValidRgn (goodRgn: RgnHandle);
```

`goodRgn` A region, in local coordinates, to be removed from a window's update region.

DESCRIPTION

The `ValidRgn` procedure removes a specified region from the update region of the window whose graphics port is the current port. Specify the region in local coordinates. The Window Manager clips it, if necessary, to fit in the window's content region.

Setting and Retrieving Other Window Characteristics

This section describes the routines that let you set and retrieve less commonly used fields in the window record.

SetWindowPic

Use the `SetWindowPic` procedure to establish a picture that the Window Manager can draw in a window's content region.

```
PROCEDURE SetWindowPic (theWindow: WindowPtr;
                        Pic: PicHandle);
```

`theWindow` A pointer to a window's window record.

`Pic` A handle to the picture to be drawn in the window.

DESCRIPTION

The `SetWindowPic` procedure stores in a window's window record a handle to a picture to be drawn in the window. When the window's content region must be updated, the Window Manager then draws the picture or part of the picture, as necessary, instead of generating an update event.

Note
The `CloseWindow` and `DisposeWindow` procedures assume that any picture pointed to by the window record field `windowPic` is stored as data, not as a resource. If your application uses a picture stored as a resource, you must release the memory it occupies by calling the Resource Manager's `ReleaseResource` procedure and set the `WindowPic` field to `NIL` before you close the window. ◆

GetWindowPic

Use the `GetWindowPic` function to retrieve a handle to a window's picture.

```
FUNCTION GetWindowPic (theWindow: WindowPtr): PicHandle;
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWindowPic` function returns a handle to the picture to be drawn in a specified window's content region. The handle must have been stored previously with the `SetWindowPic` procedure.

SetWRefCon

Use the `SetWRefCon` procedure to set the `refCon` field of a window record.

```
PROCEDURE SetWRefCon (theWindow: WindowPtr; data: LongInt);
```

`theWindow` A pointer to the window's window record.
`data` The data to be placed in the `refCon` field.

DESCRIPTION

The `SetWRefCon` procedure places the specified data in the `refCon` field of the specified window record. The `refCon` field is available to your application for any window-related data it needs to store.

SEE ALSO

See Listing 4-3 on page 4-28 for an example that sets the `refCon` field. See Listing 4-16 on page 4-60 for an example that uses the contents of the `refCon` field.

GetWRefCon

Use the `GetWRefCon` function to retrieve the reference constant from a window's window record.

```
FUNCTION GetWRefCon (theWindow: WindowPtr): LongInt;
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWRefCon` function returns the long integer data stored in the `refCon` field of the specified window record.

SEE ALSO

See the section "Managing Multiple Windows" beginning on page 4-23 for suggested ways to use the `refCon` field. See Listing 4-1 on page 4-25 for an example of an application-defined routine that gets the `refCon` field.

GetWVariant

Use the `GetWVariant` function to retrieve a window's variation code.

```
FUNCTION GetWVariant (theWindow: WindowPtr): Integer;
```

`theWindow` A pointer to the window's window record.

DESCRIPTION

The `GetWVariant` function returns the variation code of the specified window. Depending on the window's window definition function, the result of `GetWVariant` can represent one of the standard window types listed in the section "Creating a Window" beginning on page 4-25 or a variation code defined by your own window definition function.

SEE ALSO

See "Types of Windows" beginning on page 4-8 for a definition of variation codes. See "The Window Definition Function" beginning on page 4-120 for a detailed description of variation codes.

Manipulating the Desktop

This section describes the routines that let your application retrieve information about the desktop and set the desktop pattern. Ordinarily, your application doesn't need to manipulate any part of the desktop outside of its own windows.

SetDeskCPat

Use the `SetDeskCPat` procedure to set the desktop pattern on a computer that supports Color QuickDraw.

```
PROCEDURE SetDeskCPat (deskPixPat: PixPatHandle);
```

`deskPixPat` A handle to a pixel pattern.

DESCRIPTION

The `SetDeskCPat` procedure sets the desktop pattern to a specified pixel pattern, which can be drawn in more than two colors. After a call to `SetDeskCPat`, the desktop is automatically redrawn in the new pattern. If the specified pattern is a binary pattern (with a pattern type of 0), it is drawn is the current foreground and background colors. If the value of the `deskPixPat` parameter is `NIL`, `SetDeskCPat` uses the standard binary desk pattern (that is, the `'ppat'` resource with resource ID 16).

Note
For compatibility with other Macintosh applications and the system software, applications should ordinarily not change the desktop pattern. ◆

The Window Manager's desktop-painting routines can paint the desktop either in the binary pattern stored in the global variable `DeskPattern` or in a new pixel pattern. The desktop pattern used at startup is determined by the value of the parameter-RAM bit flag called `pCDeskPat`. If the value of `pCDeskPat` is 0, the Window Manager uses the new pixel pattern; if not, it uses the binary pattern stored in `DeskPattern`. The user can change the color pattern through the General Controls panel, which changes the value of `pCDeskPat`.

GetGrayRgn

Use the `GetGrayRgn` function to retrieve a handle to the current desktop region.

```
FUNCTION GetGrayRgn: RgnHandle;
```

DESCRIPTION

The `GetGrayRgn` function returns a handle to the current desktop region from the global variable `GrayRgn`.

The desktop region represents all available screen space, that is, the desktop area displayed by all monitors attached to the computer. Ordinarily, your application doesn't need to access the desktop region directly.

When your application calls `DragWindow` to let the user drag a window, it can use `GetGrayRgn` to set the limiting rectangle to the entire desktop area.

SEE ALSO

See Listing 4-9 on page 4-44 for an example that uses `GetGrayRgn` to specify the limiting rectangle when calling `DragWindow` to let the user move a window.

GetCWMgrPort

Use the `GetCWMgrPort` procedure to retrieve a pointer to the Window Manager port on a system that supports Color QuickDraw.

```
PROCEDURE GetCWMgrPort (VAR wMgrCPort: CGrafPtr);
```

`wMgrCPort` A parameter in which `GetCWMgrPort` returns a pointer to the Window Manager port.

DESCRIPTION

The `GetCWMgrPort` procedure places a pointer to the color Window Manager port in the parameter `wMgrCPort`. The `GetCWMgrPort` procedure is available only on computers with Color QuickDraw.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note
Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. ◆

GetWMgrPort

Use the `GetWMgrPort` procedure to retrieve a pointer to the Window Manager port on a system with only the original monochrome QuickDraw.

```
PROCEDURE GetWMgrPort (VAR wPort: GrafPtr);
```

`wPort` A parameter in which `GetWMgrPort` returns a pointer to the Window Manager port.

DESCRIPTION

The `GetWMgrPort` procedure places a pointer to the Window Manager port in the parameter `wPort`.

The Window Manager port is a graphics port that occupies all of the main screen. Ordinarily, your application doesn't need to access the Window Manager port.

Note
Do not change any regions of the Window Manager port. If you do, the Window Manager might not handle overlapping windows properly. ◆

Manipulating Window Color Information

This section describes the routines you use for setting and retrieving window color information. Your application does not normally change window color information.

SetWinColor

Use the `SetWinColor` procedure to set a window's window color table.

```
PROCEDURE SetWinColor (theWindow: WindowPtr;
                       newColorTable: WCTabHandle);
```

`theWindow` A pointer to the window's window record.

`newColorTable`
A handle to a window color table record, which defines the colors for the window's new color table.

DESCRIPTION

The `SetWinColor` procedure sets a window's color table. If the window has no auxiliary window record, it creates a new one with the specified window color table and adds it to the auxiliary window list. If the window already has an auxiliary record, its window color table is replaced. The Window Manager then redraws the window frame and highlighted text in the new colors and sets the window's background color to the new content color.

If the new color table has the same entries as the default color table, `SetWinColor` changes the auxiliary window record so that it points to the default color table.

Window color table resources (resources of type `'wctb'`) should not be purgeable.

If you specify a value of `NIL` for the parameter `theWindow`, `SetWinColor` changes the default color table in memory. Your application shouldn't, however, change the default color table.

SEE ALSO

For a description of a window color table, see "The Window Color Table Record" on page 4-71. For a description of the auxiliary window record, see "The Auxiliary Window Record" on page 4-73. For a description of the `'wctb'` resource, see "The Window Color Table Resource" on page 4-127.

GetAuxWin

Use the `GetAuxWin` function to retrieve a handle to a window's auxiliary window record.

```
FUNCTION GetAuxWin (theWindow: WindowPtr;
                    VAR awHndl: AuxWinHandle): Boolean;
```

`theWindow` A pointer to the window's window record.

`awHndl` A handle to the window's auxiliary window record.

DESCRIPTION

The `GetAuxWin` function returns a Boolean value that reports whether or not the window has an auxiliary window record, and it sets the variable parameter `awHndl` to the window's auxiliary window record.

If the window has no auxiliary window record, `GetAuxWin` places the default window color table in `awHndl` and returns a value of `FALSE`.

SEE ALSO

For a description of the auxiliary window record, see "The Auxiliary Window Record" on page 4-73.

Low-Level Routines

This section describes the low-level routines that are called by higher-level Window Manager routines. Ordinarily, you won't need to use these routines.

CheckUpdate

The Event Manager uses the `CheckUpdate` function to scan the window list for windows that need updating.

```
FUNCTION CheckUpdate (VAR theEvent: EventRecord): Boolean;
```

`theEvent` An event record to be filled in if a window needs updating.

DESCRIPTION

The `CheckUpdate` function scans the window list from front to back, checking for a visible window that needs updating (that is, a visible window whose update region is not empty). If it finds one whose window record contains a picture handle, it redraws the window itself and continues through the list. If it finds a window record whose update region is not empty and whose window record does not contain a picture handle, it stores an update event in the parameter `theEvent` and returns `TRUE`. If it finds no such window, it returns `FALSE`.

The Event Manager is the only software that ordinarily calls `CheckUpdate`.

ClipAbove

The Window Manager uses the `ClipAbove` procedure to determine the clip region of the Window Manager port for displaying a window.

```
PROCEDURE ClipAbove (window: WindowPeek);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `ClipAbove` procedure sets the clip region of the Window Manager port to be the area of the desktop that intersects the current clip region, minus the structure regions of all the windows in front of the specified window.

The ClipAbove procedure retrieves the desktop region from the global variable GrayRgn.

SaveOld

The Window Manager uses the SaveOld procedure to save a window's current structure and content regions preparatory to updating the window.

```
PROCEDURE SaveOld (window: WindowPeek);
```

window A pointer to the window's complete window record.

DESCRIPTION

The SaveOld procedure saves the specified window's current structure region and content region for the DrawNew procedure. Each call to SaveOld must be balanced by a subsequent call to DrawNew.

DrawNew

The Window Manager uses the DrawNew procedure to erase and update changed window regions.

```
PROCEDURE DrawNew (window: WindowPeek; update: Boolean);
```

window A pointer to the window's complete window record.

update A Boolean value that determines whether the regions are updated.

DESCRIPTION

The DrawNew procedure erases the parts of a window's structure and content regions that are part of the window's former state and part of its new state but not both. That is,

```
(OldStructure XOR NewStructure) UNION (OldContent XOR NewContent)
```

If the update parameter is set to TRUE, DrawNew also updates the erased regions.

▲ **WARNING**
In Pascal, SaveOld and DrawNew are not nestable. ▲

ASSEMBLY-LANGUAGE INFORMATION

In assembly language, you can nest SaveOld and DrawNew if you save and restore the values of the global variables OldStructure and OldContent.

PaintOne

The Window Manager uses the `PaintOne` procedure to redraw the invalid, exposed portions of one window on the desktop.

```
PROCEDURE PaintOne (window: WindowPeek; clobberedRgn: RgnHandle);
```

`window` A pointer to the window's complete window record.

`clobberedRgn`
A handle to the region that has become invalid.

DESCRIPTION

The `PaintOne` procedure "paints" the invalid portion of the specified window and all windows above it. It draws as much of the window frame as is in `clobberedRgn` and, if some content region is exposed, erases the exposed area (paints it with the background pattern) and adds it to the window's update region. If the value of the `window` parameter is `NIL`, the window is the desktop, and `PaintOne` paints it with the desktop pattern.

ASSEMBLY-LANGUAGE INFORMATION

The global variables `SaveUpdate` and `PaintWhite` are flags used by `PaintOne`. Normally both flags are set. Clearing `SaveUpdate` prevents `clobberedRgn` from being added to the window's update region. Clearing `PaintWhite` prevents `clobberedRgn` from being erased before being added to the update region (this is useful, for example, if the background pattern of the window isn't the background pattern of the desktop). The Window Manager sets both flags periodically, so you should clear the appropriate flags each time you need them to be clear.

PaintBehind

The Window Manager uses the `PaintBehind` procedure to redraw a series of windows in the window list.

```
PROCEDURE PaintBehind (startWindow: WindowPeek;
                       clobberedRgn: RgnHandle);
```

`startWindow`
A pointer to the window's complete window record.

`clobberedRgn`
A handle to the region that has become invalid.

DESCRIPTION

The `PaintBehind` procedure calls `PaintOne` for `startWindow` and all the windows behind `startWindow`, clipped to `clobberedRgn`.

ASSEMBLY-LANGUAGE INFORMATION

Because `PaintBehind` clears the global variable `PaintWhite` before calling `PaintOne`, `clobberedRgn` isn't erased. The `PaintWhite` global variable is reset after the call to `PaintOne`.

CalcVis

The Window Manager uses the `CalcVis` procedure to calculate the visible region of a window.

```
PROCEDURE CalcVis (window: WindowPeek);
```

`window` A pointer to the window's complete window record.

DESCRIPTION

The `CalcVis` procedure calculates the visible region of the specified window by starting with its content region and subtracting the structure region of each window in front of it.

CalcVisBehind

The Window Manager uses the `CalcVisBehind` procedure to calculate the visible regions of a series of windows.

```
PROCEDURE CalcVisBehind (startWindow: WindowPeek;
                         clobberedRgn: RgnHandle);
```

`startWindow`
A pointer to a window's window record.

`clobberedRgn`
A handle to the desktop region that has become invalid.

DESCRIPTION

The `CalcVisBehind` procedure calculates the visible regions of the window specified by the `startWindow` parameter and all windows behind `startWindow` that intersect `clobberedRgn`. It is called after `PaintBehind`.

Application-Defined Routine

This section describes the window definition function. The Window Manager supplies window definition functions that handle the standard window types described in "Types of Windows" beginning on page 4-8.

The Window Definition Function

If your application defines its own window types, you must supply your own window definition function to handle them. Store your definition function as a resource of type `'WDEF'` with an ID from 128 through 4096. (Window definition function resource IDs 0 and 1 are the default window definition functions; resource IDs 2 through 127 are reserved by Apple Computer, Inc.)

Your window definition function can support up to 16 variation codes, which are identified by integers 0 through 15. To invoke your own window type, you specify the window's definition ID, which contains the resource ID of the window's definition function in the upper 12 bits and the variation code in the lower 4 bits. Thus, for a given resource ID and variation code, the window definition ID is

(16 * resource ID) + (variation code)

When you create a window, the Window Manager calls the Resource Manager to access the window definition function. The Resource Manager reads the window definition function into memory and returns a handle to it. The Window Manager stores this handle in the `windowDefProc` field of the window record. (If 24-bit addressing is in effect, the Window Manager stores the variation code in the lower 4 bits of the `windowDefProc` field; if 32-bit addressing is in effect, the Window Manager stores the variation code elsewhere.) Later, when it needs to perform a type-dependent action on the window, the Window Manager calls the window definition function and passes it the variation code as a parameter.

MyWindow

The window definition function is responsible for drawing the window frame, reporting the region where mouse-down events occur, calculating the window's structure region and content region, drawing the size box, resizing the window frame when the user drags the size box, and performing any customized initialization or disposal tasks.

You can give your window definition function any name you wish. It takes four parameters and returns a result code:

```
FUNCTION MyWindow (varCode: Integer; theWindow: WindowPtr;
                   message: Integer; param: LongInt): LongInt;
```

`varCode` The window's variation code.

`theWindow` A pointer to the window's window record.

`message` A code for the task to be performed. The `message` parameter has one of these values:

```
CONST
   wDraw       = 0;  {draw window frame}
   wHit        = 1;  {report where mouse-down event }
                     { occurred}
   wCalcRgns   = 2;  {calculate strucRgn and contRgn}
   wNew        = 3;  {perform additional }
                     { initialization}
   wDispose    = 4;  {perform additional disposal }
                     { tasks}
   wGrow       = 5;  {draw grow image during resizing}
   wDrawGIcon  = 6;  {draw size box and scroll bar }
                     { outline}
```

The subsections that follow explain each of these tasks in detail.

`param` Data associated with the task specified by the `message` parameter. If the task requires no data, this parameter is ignored.

Your window definition function performs whatever task is specified by the `message` parameter and returns a function result if appropriate. If the task performed requires no result code, return 0.

The function's entry point must be at the beginning of the function.

You can set up the various tasks as subroutines inside the window definition function, but you're not required to do so.

Drawing the Window Frame

When you receive a `wDraw` message, draw the window frame in the current graphics port, which is the Window Manager port.

You must make certain checks to determine exactly how to draw the frame. If the value of the `visible` field in the window record is `FALSE`, you should do nothing; otherwise, you should examine the `param` parameter and the status flags in the window record:

- If the value of `param` is 0, draw the entire window frame.
- If the value of `param` is 0 and the `hilited` field in the window record is `TRUE`, highlight the frame to show that the window is active.
  - If the value of the `goAwayFlag` field in the window record is also `TRUE`, draw a close box in the window frame.
  - If the value of the `spareFlag` field in the window record is also `TRUE`, draw a zoom box in the window frame.
- If the value of the `param` parameter is `wInGoAway`, add highlighting to, or remove it from, the window's close box. Figure 4-19 on page 4-46 illustrates the close box with and without highlighting as drawn by the Window Manager's window definition function.

- If the value of the `param` parameter is `wInZoom`, add highlighting to, or remove it from, the window's zoom box. Figure 4-20 on page 4-47 illustrates the zoom box with and without highlighting as drawn by the Window Manager's window definition function.

Note

When the Window Manager calls a window definition function with a message of `wDraw`, it stores a value of type `Integer` in the `param` parameter without clearing the high-order word. When processing the `wDraw` message, use only the low-order word of the `param` parameter. ◆

The window frame typically but not necessarily includes the window's title, which should be displayed in the system font and system font size. The Window Manager port is already set to use the system font and system font size.

When designing a title bar that includes the window title, allow at least 16 pixels vertically to support localization for script systems in which the system font can be no smaller than 12 points.

Note

Nothing drawn outside the window's structure region is visible. ◆

Returning the Region of a Mouse-Down Event

When you receive a `wHit` message, you must determine where the cursor was when the mouse button was pressed. The `wHit` message is accompanied by the mouse location, in global coordinates, in the `param` parameter. The vertical coordinate is in the high-order word of the parameter, and the horizontal coordinate is in the low-order word. You return one of these constants:

```
CONST
   wNoHit      = 0;  {none of the following}
   wInContent  = 1;  {in content region (except grow, if active)}
   wInDrag     = 2;  {in drag region}
   wInGrow     = 3;  {in grow region (active window only)}
   wInGoAway   = 4;  {in go-away region (active window only)}
   wInZoomIn   = 5;  {in zoom box for zooming in (active window }
                     { only)}
   wInZoomOut  = 6;  {in zoom box for zooming out (active window }
                     { only)}
```

The return value `wNoHit` might mean (but not necessarily) that the point isn't in the window. The standard window definition functions, for example, return `wNoHit` if the point is in the window frame but not in the title bar.

Return the constants `wInGrow`, `wInGoAway`, `wInZoomIn`, and `wInZoomOut` only if the window is active—by convention, the size box, close box, and zoom box aren't drawn if the window is inactive. In an inactive document window, for example, a mouse-down event in the part of the title bar that would contain the close box if the window were active is reported as `wInDrag`.

Calculating Regions

When you receive the `wCalcRgns` message, you calculate the window's structure and content regions based on the current graphics port's port rectangle. These regions, whose handles are in the `strucRgn` and `contRgn` fields of the window record, are in global coordinates. The Window Manager requests this operation only if the window is visible.

▲ **WARNING**
When you calculate regions for your own type of window, do not alter the clip region or the visible region of the window's graphics port. The Window Manager and QuickDraw take care of this for you. Altering the clip region or visible region may damage other windows. ▲

Initializing a New Window

When you receive the `wNew` message, you can perform any type-specific initialization that may be required. If the content region has an unusual shape, for example, you might allocate memory for the region and store the region handle in the `dataHandle` field of the window record. The initialization routine for a standard document window creates the `wStateData` record for storing zooming data.

Disposing of a Window

When you receive the `wDispose` message, you can perform any additional tasks necessary for disposing of a window. You might, for example, release memory that was allocated by the initialization routine. The dispose routine for a standard document window disposes of the `wStateData` record.

Resizing a Window

When you receive the `wGrow` message, draw a grow image of the window. With the `wGrow` message you receive a pointer to a rectangle, in global coordinates, whose upper-left corner is aligned with the port rectangle of the window's graphics port. Your grow image should fit inside the rectangle. As the user drags the mouse, the Window Manager sends repeated `wGrow` messages, so that you can change your grow image to match the changing mouse location.

Draw the grow image in the current graphics port, which is the Window Manager port, in the current pen pattern and pen mode. These are set up (as `gray` and `notPatXor`) to conform to the Macintosh user interface guidelines.

The grow routine for a standard document window draws a dotted (gray) outline of the window and also the lines delimiting the title bar, size box, and scroll bar areas.

Drawing the Size Box

When you receive the `wDrawGIcon` message, you draw the size box in the content region if the window is active—if the window is inactive, draw whatever is appropriate to show that the window cannot currently be sized.

Note
If the size box is located in the window frame instead of the content region, do nothing in response to the `wDrawGIcon` message, instead drawing the size box in response to the `wDraw` message. ◆

The routine that draws a size box for an active document window draws the size box in the lower-right corner of the port rectangle of the window's graphics port. It also draws lines delimiting the size box and scroll bar areas. For an inactive document window, it erases the size box and draws the delimiting lines.

Resources

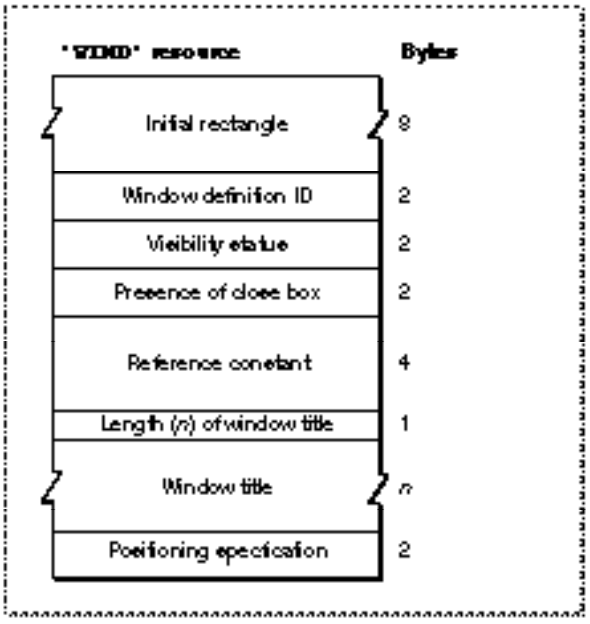
This section describes the resources used by the Window Manager:

- the `'WIND'` resource, used for describing the characteristics of windows
- the `'WDEF'` resource, which holds a window definition function
- the `'wctb'` resource, which defines the colors to be used for a window's frame and highlighting

The Window Resource

You typically define a window resource for each type of window that your application creates. Figure 4-24 illustrates a compiled `'WIND'` resource.

Figure 4-24 Structure of a compiled window (`'WIND'`) resource



A compiled version of a window resource contains the vollowing elements:

- The upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional positioning code described later in this section.
- The window's definition ID, which incorporates both the resource ID of the window definition function that will handle the window and an optional variation code. Together, the window definition function resource ID and the variation code define a window type. Place the resource ID of the window definition function in the upper 12 bits of the definition ID. Window definition functions with IDs 0 through 127 are reserved for use by Apple Computer, Inc. Place the optional variation code in the lower 4 bits of the definition ID.

  If you're using one of the standard window types (described in "Types of Windows" beginning on page 4-8), the definition ID is one of the window-type constants:

```
CONST
documentProc      = 0;  {movable, sizable window, }
                        { no zoom box}
dBoxProc          = 1;  {alert box or modal dialog box}
plainDBox         = 2;  {plain box}
altDBoxProc       = 3;  {plain box with shadow}
noGrowDocProc     = 4;  {movable window, no size box or }
                        { zoom box}
movableDBoxProc   = 5;  {movable modal dialog box}
zoomDocProc       = 8;  {standard document window}
zoomNoGrow        = 12; {zoomable, nonresizable window}
rDocProc          = 16; {rounded-corner window}
```

  You can also add a zoom box to a movable modal dialog box by specifying the sum of two constants: `movableDBoxProc + zoomDocProc`, but a zoom box is not recommended on any dialog box.

  You can control the angle of curvature on a rounded-corner window (window type `rDocProc`) by adding one of these integers:

Window definition ID	Diameters of curvature
`rDocProc`	16, 16
`rDocProc + 2`	4, 4
`rDocProc + 4`	6, 6
`rDocProc + 6`	10, 10

- A specification that determines whether the window is visible or invisible. This characteristic controls only whether the Window Manager displays the window, not necessarily whether the window can be seen on the screen. (A visible window entirely covered by other windows, for example, is "visible" even though the user cannot see it.) You typically create a new window in an invisible state, build the content area of the window, and then display the completed window.

- A specification that determines whether or not the window has a close box. The Window Manager draws the close box when it draws the window frame. The window type specified in the second field determines whether a window can support a close box; this field determines whether the close box is present.
- A reference constant, which your application can use for whatever data it needs to store. When it builds a new window record, the Window Manager stores, in the `refCon` field, whatever value you specify in the fifth element of the window resource. You can also put a placeholder here and then set the `refCon` field yourself with the `SetWRefCon` procedure.
- A string that specifies the window title. The first byte of the string specifies the length of the string (that is, the number of characters in the title plus 1 byte for the length), in bytes.
- An optional positioning specification that overrides the window position established by the rectangle in the first field. The positioning value can be one of the integers defined by the constants listed here. In these constant names, the terms have the following meanings:

`center`	Centered both horizontally and vertically, relative either to a screen or to another window (if a window to be centered relative to another window is wider than the window that preceded it, it is pinned to the left edge; a narrower window is centered)
`stagger`	Located 10 pixels to the right and 10 pixels below the upper-left corner of the last window (in the case of staggering relative to a screen, the first window is placed just below the menu bar at the left edge of the screen, and subsequent windows are placed on that screen relative to the first window)
`alert position`	Centered horizontally and placed in the "alert position" vertically, that is, with about one-fifth of the window or screen above the new window and the rest below
`parent window`	The window in which the user was last working

The seventh element of the resource can contain one of the values specified by these constants:

```
CONST noAutoCenter            = 0x0000;{use initial }
                                       { location}
      centerMainScreen        = 0x280A;{center on main }
                                       { screen}
      alertPositionMainScreen = 0x300A;{place in alert }
                                       { position on main }
                                       { screen}
      staggerMainScreen       = 0x380A;{stagger on main }
                                       { screen}
      centerParentWindow      = 0xA80A;{center on parent }
                                       { window}
```

```
alertPositionParentWindow  = 0xB00A;{place in alert }
                                    { position on }
                                    { parent window}
staggerParentWindow        = 0xB80A;{stagger relative }
                                    { to parent window}
centerParentWindowScreen   = 0x680A;{center on parent }
                                    { window screen}
alertPositionParentWindowScreen
                           = 0x700A;{place in alert }
                                    { position on }
                                    { parent window }
                                    { screen}
staggerParentWindowScreen  = 0x780A;{stagger on parent }
                                    { window screen}
```

The positioning constants are convenient when the user is creating new documents or when you are handling your own dialog boxes and alert boxes. When you are creating a new window to display a previously saved document, however, you should display the new window in the same rectangle as the previous window (that is, the window the document occupied when it was last saved). For more information, see "Positioning a Document Window on the Desktop" beginning on page 4-30.

Use the `GetNewCWindow` or `GetNewWindow` function to read a `'WIND'` resource. Both functions create a new window record and fill it in according to the values specified in a `'WIND'` resource.

The Window Definition Function Resource

Window definition functions are stored as resources of type `'WDEF'`. The `'WDEF'` resource is simply the executable code for the window definition function.

The two standard window definition functions supplied by the Window Manager use resource IDs 0 and 1.

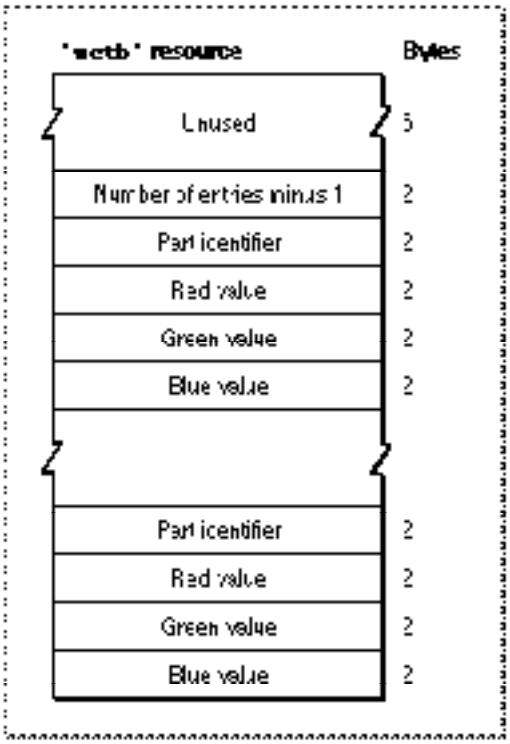
The Window Color Table Resource

You can specify your own window color tables as resources of type `'wctb'`.

Ordinarily, you should not define your own window color tables, unless you have some extraordinary need to control the color of a window's frame or text highlighting. To assign a table to a window when you create the window, provide a window color table (`'wctb'`) resource with the same resource ID as the `'WIND'` resource from which you create the window.

The window color table resource is an exact image of the window color table data structure. Figure 4-25 illustrates the contents of a compiled `'wctb'` resource.

Figure 4-25 Structure of a compiled window color table (`'wctb'`) resource



A compiled version of a window resource contains the following elements:

- An unused field 6 bytes long.
- An integer that specifies the number of entries in the resource (that is, the number of color specification records) minus 1.
- A series of color specification records, each of which consists of a 2-byte part identifier and three 2-byte color values. The part identifier is an integer specified by one of these constants:

```
CONST wContentColor     = 0;  {content region background}
      wFrameColor       = 1;  {window frame}
      wTextColor        = 2;  {window title and button text}
      wHiliteColor      = 3;  {reserved}
      wTitleBarColor    = 4;  {reserved}
      wHiliteColorLight = 5;  {lightest stripes in title bar }
                              { and lightest dimmed text}
      wHiliteColorDark  = 6;  {darkest stripes in title bar }
                              { and darkest dimmed text}
      wTitleBarLight    = 7;  {lightest parts of title bar }
                              { background}
```

```
wTitleBarDark     = 8;  {darkest parts of title bar }
                        { background}
wDialogLight      = 9;  {lightest element of dialog box }
                        { frame}
wDialogDark       = 10; {darkest element of dialog box }
                        { frame}
wTingeLight       = 11; {lightest window tinging}
wTingeDark        = 12; {darkest window tinging}
```

The color values are simply the intensity of the red, green, and blue in each window part (see *Inside Macintosh: Imaging* for a description of RGB color).

Summary of the Window Manager

Pascal Summary

Constants

```
CONST
   {window types}
   documentProc      = 0;  {movable, sizable window, no zoom box}
   dBoxProc          = 1;  {alert box or modal dialog box}
   plainDBox         = 2;  {plain box}
   altDBoxProc       = 3;  {plain box with shadow}
   noGrowDocProc     = 4;  {movable window, no size box or }
                           { zoom box}
   movableDBoxProc   = 5;  {movable modal dialog box}
   zoomDocProc       = 8;  {standard document window}
   zoomNoGrow        = 12; {zoomable, nonresizable window}
   rDocProc          = 16; {rounded-corner window}

   {window kinds}
   dialogKind        = 2;  {dialog or alert box window}
   userKind          = 8;  {window created by the application}

   {part codes returned by FindWindow}
   inDesk            = 0;  {none of the following}
   inMenuBar         = 1;  {in menu bar}
   inSysWindow       = 2;  {in desk accessory window}
   inContent         = 3;  {anywhere in content region except size }
                           { box if window is active, }
                           { anywhere including size box if window }
                           { is inactive}
   inDrag            = 4;  {in drag (title bar) region}
   inGrow            = 5;  {in size box (active window only)}
   inGoAway          = 6;  {in close box}
   inZoomIn          = 7;  {in zoom box (window in standard state)}
   inZoomOut         = 8;  {in zoom box (window in user state)}

   {axis constraints on DragGrayRgn}
   noConstraint      = 0;  {no constraints}
   hAxisOnly         = 1;  {move on horizontal axis only}
   vAxisOnly         = 2;  {move on vertical axis only}
```

```
{window definition function task codes}
wDraw       = 0;  {draw window frame}
wHit        = 1;  {report where mouse-down occurred}
wCalcRgns   = 2;  {calculate strucRgn and contRgn}
wNew        = 3;  {perform additional initialization}
wDispose    = 4;  {perform additional disposal tasks}
wGrow       = 5;  {draw grow image during resizing}
wDrawGIcon  = 6;  {draw size box and scroll bar outline}

{window definition function wHit return codes}
wNoHit      = 0;  {none of the following}
wInContent  = 1;  {anywhere in content region except size }
                  { box if window is active, }
                  { anywhere including size box if window }
                  { is inactive}
wInDrag     = 2;  {in drag (title bar) region}
wInGrow     = 3;  {in size box (active window only)}
wInGoAway   = 4;  {in close box}
wInZoomIn   = 5;  {in zoom box (window in standard state)}
wInZoomOut  = 6;  {in zoom box (window in user state)}

{window color information table part codes}
wContentColor     = 0;     {content region background}
wFrameColor       = 1;     {window outline}
wTextColor        = 2;     {window title and button text}
wHiliteColor      = 3;     {reserved}
wTitleBarColor    = 4;     {reserved}
wHiliteColorLight = 5;     {lightest stripes in title bar }
                           { and lightest dimmed text}
wHiliteColorDark  = 6;     {darkest stripes in title bar }
                           { and darkest dimmed text}
wTitleBarLight    = 7;     {lightest parts of title bar background}
wTitleBarDark     = 8;     {darkest parts of title bar background}
wDialogLight      = 9;     {lightest element of dialog box frame}
wDialogDark       = 10;    {darkest element of dialog box frame}
wTingeLight       = 11;    {lightest window tinging}
wTingeDark        = 12;    {darkest window tinging}

{resource ID of desktop pattern}
deskPatID         = 16;
```

Data Types

```
TYPE  CWindowPtr  = CGrafPtr;
      CWindowPeek = ^CWindowRecord;

CWindowRecord =
RECORD
   port:          CGrafPort;     {window's graphics port}
   windowKind:    Integer;       {class of window}
   visible:       Boolean;       {visibility}
   hilited:       Boolean;       {highlighting}
   goAwayFlag:    Boolean;       {presence of close box}
   spareFlag:     Boolean;       {presence of zoom box}
   strucRgn:      RgnHandle;     {handle to structure region}
   contRgn:       RgnHandle;     {handle to content region}
   updateRgn:     RgnHandle;     {handle to update region}
   windowDefProc: Handle;        {handle to window definition function}
   dataHandle:    Handle;        {handle to window state data record}
   titleHandle:   StringHandle;  {handle to window title}
   titleWidth:    Integer;       {title width in pixels}
   controlList:   ControlHandle; {handle to control list}
   nextWindow:    CWindowPeek;   {pointer to next window record in }
                                 { window list}
   windowPic:     PicHandle;     {handle to optional picture}
   refCon:        LongInt;       {storage available to your application}
END;

WindowPtr   = GrafPtr;
WindowPeek  = ^WindowRecord;

WindowRecord =
RECORD                           {all fields have same use as }
                                 { in color window record}
   port:          GrafPort;      {window's graphics port}
   windowKind:    Integer;       {class of window}
   visible:       Boolean;       {visibility}
   hilited:       Boolean;       {highlighting}
   goAwayFlag:    Boolean;       {presence of close box}
   spareFlag:     Boolean;       {presence of zoom box}
   strucRgn:      RgnHandle;     {handle to structure region}
   contRgn:       RgnHandle;     {handle to content region}
   updateRgn:     RgnHandle;     {handle to update region}
   windowDefProc: Handle;        {handle to window definition function}
   dataHandle:    Handle;        {handle to window state data record}
```

```
   titleHandle:   StringHandle;  {handle to window title}
   titleWidth:    Integer;       {title width in pixels}
   controlList:   ControlHandle; {handle to control list}
   nextWindow:    WindowPeek;    {pointer to next window record in }
                                 { window list}
   windowPic:     PicHandle;     {handle to optional picture}
   refCon:        LongInt;       {storage available to your application}
END;

WStateDataPtr = ^WStateData;
WStateDataHandle = ^WStateDataPtr;

WStateData =            {zoom state data record}
RECORD
   userState:  Rect;    {size and location established by user}
   stdState:   Rect;    {size and location established by application}
END;

WCTabPtr = ^WinCTab;
WCTabHandle = ^WCTabPtr;

WinCTab =                       {window color information table}
RECORD
   wCSeed:     LongInt;         {reserved}
   wCReserved: Integer;         {reserved}
   ctSize:     Integer;         {number of entries in table -1}
   ctTable:    ARRAY [0..4] OF ColorSpec;
                                {array of color specification records}
END;

ColorSpec   =
RECORD
   value:      Integer;         {part identifier}
   rgb:        RGBColor;        {RGB value}
END;

AuxWinHandle= ^AuxWinPtr;
AuxWinPtr   = ^AuxWinRec;

AuxWinRec   =                   {auxiliary window record}
RECORD
   awNext:        AuxWinHandle;  {handle to next record}
   awOwner:       WindowPtr;     {pointer to window}
   awCTable:      CTabHandle;    {handle to color table}
   dialogCItem:   Handle;        {storage used by Dialog Manager}
```

```
   awFlags:       LongInt;       {reserved}
   awReserved:    CTabHandle;    {reserved}
   awRefCon:      LongInt;       {reference constant, for }
                                 { use by application}
END;
```

Window Manager Routines

Initializing the Window Manager

```
PROCEDURE InitWindows;
```

Creating Windows

```
FUNCTION GetNewCWindow      (windowID: Integer; wStorage: Ptr;
                             behind: WindowPtr): WindowPtr;
FUNCTION GetNewWindow       (windowID: Integer; wStorage: Ptr;
                             behind: WindowPtr): WindowPtr;
FUNCTION NewCWindow         (wStorage: Ptr; boundsRect: Rect;
                             title: Str255; visible: Boolean;
                             procID: Integer; behind: WindowPtr;
                             goAwayFlag: Boolean;
                             refCon: LongInt): WindowPtr;
FUNCTION NewWindow          (wStorage: Ptr; boundsRect: Rect;
                             title: Str255; visible: Boolean;
                             theProc: Integer; behind: WindowPtr;
                             goAwayFlag: Boolean;
                             refCon: LongInt): WindowPtr;
```

Naming Windows

```
PROCEDURE SetWTitle         (theWindow: WindowPtr; title: Str255);
PROCEDURE GetWTitle         (theWindow: WindowPtr; VAR title: Str255);
```

Displaying Windows

```
PROCEDURE DrawGrowIcon      (theWindow: WindowPtr);
PROCEDURE SelectWindow      (theWindow: WindowPtr);
PROCEDURE ShowWindow        (theWindow: WindowPtr);
PROCEDURE HideWindow        (theWindow: WindowPtr);
PROCEDURE ShowHide          (theWindow: WindowPtr; showFlag: Boolean);
PROCEDURE HiliteWindow      (theWindow: WindowPtr; fHilite: Boolean);
PROCEDURE BringToFront      (theWindow: WindowPtr);
PROCEDURE SendBehind        (theWindow, behindWindow: WindowPtr);
```

Retrieving Window Information

```
FUNCTION FindWindow         (thePoint: Point;
                             VAR theWindow: WindowPtr): Integer;
FUNCTION FrontWindow        : WindowPtr;
```

Moving Windows

```
PROCEDURE DragWindow        (theWindow: WindowPtr;
                             startPt: Point; boundsRect: Rect);
PROCEDURE MoveWindow        (theWindow: WindowPtr;
                             hGlobal, vGlobal: Integer; front: Boolean);
FUNCTION DragGrayRgn        (theRgn: RgnHandle; startPt: Point;
                             limitRect, slopRect: Rect; axis: Integer;
                             actionProc: ProcPtr): LongInt;
FUNCTION PinRect            (theRect: Rect; thePt: Point): LongInt;
```

Resizing Windows

```
FUNCTION GrowWindow         (theWindow: WindowPtr;
                             startPt: Point; sizeRect: Rect): LongInt;
PROCEDURE SizeWindow        (theWindow: WindowPtr; w, h: Integer;
                             fUpdate: Boolean);
```

Zooming Windows

```
FUNCTION TrackBox           (theWindow: WindowPtr; thePt: Point;
                             partCode: Integer): Boolean;
PROCEDURE ZoomWindow        (theWindow: WindowPtr;
                             partCode: Integer; front: Boolean);
```

Closing and Deallocating Windows

```
FUNCTION TrackGoAway        (theWindow: WindowPtr; thePt: Point): Boolean;
PROCEDURE CloseWindow       (theWindow: WindowPtr);
PROCEDURE DisposeWindow     (theWindow: WindowPtr);
```

Maintaining the Update Region

```
PROCEDURE BeginUpdate       (theWindow: WindowPtr);
PROCEDURE EndUpdate         (theWindow: WindowPtr);
PROCEDURE InvalRect         (badRect: Rect);
PROCEDURE InvalRgn          (badRgn: RgnHandle);
PROCEDURE ValidRect         (goodRect: Rect);
PROCEDURE ValidRgn          (goodRgn: RgnHandle);
```

Setting and Retrieving Other Window Characteristics

```
PROCEDURE SetWindowPic      (theWindow: WindowPtr; Pic: PicHandle);
FUNCTION GetWindowPic       (theWindow: WindowPtr): PicHandle;
PROCEDURE SetWRefCon        (theWindow: WindowPtr; data: LongInt);
FUNCTION GetWRefCon         (theWindow: WindowPtr): LongInt;
FUNCTION GetWVariant        (theWindow: WindowPtr): Integer;
```

Manipulating the Desktop

```
PROCEDURE SetDeskCPat       (deskPixPat: PixPatHandle);
FUNCTION GetGrayRgn         : RgnHandle;
PROCEDURE GetCWMgrPort      (VAR wMgrCPort: CGrafPtr);
PROCEDURE GetWMgrPort       (VAR wPort: GrafPtr);
```

Manipulating Window Color Information

```
PROCEDURE SetWinColor       (theWindow: WindowPtr;
                             newColorTable: WCTabHandle);
FUNCTION GetAuxWin          (theWindow: WindowPtr;
                             VAR awHndl: AuxWinHandle): Boolean;
```

Low-Level Routines

```
FUNCTION CheckUpdate        (VAR theEvent: EventRecord): Boolean;
PROCEDURE ClipAbove         (window: WindowPeek);
PROCEDURE SaveOld           (window: WindowPeek);
PROCEDURE DrawNew           (window: WindowPeek; update: Boolean);
PROCEDURE PaintOne          (window: WindowPeek; clobberedRgn: RgnHandle);
PROCEDURE PaintBehind       (startWindow: WindowPeek;
                             clobberedRgn: RgnHandle);
PROCEDURE CalcVis           (window: WindowPeek);
PROCEDURE CalcVisBehind     (startWindow: WindowPeek;
                             clobberedRgn: RgnHandle);
```

Application-Defined Routine

The Window Definition Function

```
FUNCTION MyWindow           (varCode: Integer; theWindow: WindowPtr;
                             message: Integer; param: LongInt): LongInt;
```

C Summary

Constants

```
enum {
   /*window types*/
   documentProc      = 0,  /*movable, sizable window, no zoom box*/
   dBoxProc          = 1,  /*alert box or modal dialog box*/
   plainDBox         = 2,  /*plain box*/
   altDBoxProc       = 3,  /*plain box with shadow*/
   noGrowDocProc     = 4,  /*movable window, no size box or zoom box*/
   movableDBoxProc   = 5,  /*movable modal dialog box*/
   zoomDocProc       = 8,  /*standard document window*/
   zoomNoGrow        = 9,  /*zoomable, nonresizable window*/
   rDocProc          = 16, /*rounded-corner window*/

   /*window kinds*/
   dialogKind        = 2,  /*dialog or alert box window*/
   userKind          = 8,  /*window created by the application*/

   /*part codes returned by FindWindow*/
   inDesk            = 0,  /*none of the following*/
   inMenuBar         = 1,  /*in menu bar*/
   inSysWindow       = 2,  /*in desk accessory window*/
   inContent         = 3,  /*anywhere in content region except size box if*/
                           /* window is active, anywhere including */
                           /* size box if window is inactive*/
   inDrag            = 4,  /*in drag (title bar) region*/
   inGrow            = 5,  /*in size box (active window only)*/
   inGoAway          = 6,  /*in close box*/
   inZoomIn          = 7,  /*in zoom box (window in standard state)*/
   inZoomOut         = 8   /*in zoom box (window in user state)*/
};

enum {
   /*axis constraints on DragGrayRgn*/
   noConstraint      = 0,  /*no constraints*/
   hAxisOnly         = 1,  /*move on horizontal axis only*/
   vAxisOnly         = 2   /*move on vertical axis only*/
};
```

```
enum {
   /*window definition function task codes*/
   wDraw       = 0,  /*draw window frame*/
   wHit        = 1,  /*report where mouse-down occurred*/
   wCalcRgns   = 2,  /*calculate strucRgn and contRgn*/
   wNew        = 3,  /*perform additional initialization*/
   wDispose    = 4,  /*perform additional disposal tasks*/
   wGrow       = 5,  /*draw grow image during resizing*/
   wDrawGIcon  = 6,  /*draw size box and scroll bar outline*/

   /*window definition function wHit return codes*/
   wNoHit      = 0,  /*none of the following*/
   wInContent  = 1,  /*in content region (except grow, if active)*/
   wInDrag     = 2,  /*in drag region*/
   wInGrow     = 3,  /*in grow region (active window only)*/
   wInGoAway   = 4,  /*in go-away region (active window only)*/
   wInZoomIn   = 5,  /*in zoom box for zooming in (active window */
                     /* only)*/
   wInZoomOut  = 6,  /*in zoom box for zooming out (active window */
                     /* only)*/
   deskPatID   = 16, /*resource ID of desktop pattern*/

   /*window color information table part codes*/
   wContentColor     = 0,     /*the background of the window's */
                              /* content region*/
   wFrameColor       = 1,     /*the window outline*/
   wTextColor        = 2,     /*window title and text in buttons*/
   wHiliteColor      = 3,     /*reserved*/
   wTitleBarColor    = 4,     /*reserved*/
   wHiliteColorLight = 5,     /*lightest stripes in title bar */
                              /* and lightest dimmed text*/
   wHiliteColorDark  = 6,     /*darkest stripes in title bar */
                              /* and darkest dimmed text*/
   wTitleBarLight    = 7,     /*lightest parts of title bar background*/
   wTitleBarDark     = 8,     /*darkest parts of title bar background*/
   wDialogLight      = 9,     /*lightest element of dialog box frame*/
   wDialogDark       = 10,    /*darkest element of dialog box frame*/
   wTingeLight       = 11,    /*lightest window tinging*/
   wTingeDark        = 12     /*darkest window tinging*/
};
```

Data Types

```
struct CWindowRecord {
   CGrafPort            port;          /*window's graphics port*/
   short                windowKind;    /*class of the window*/
   Boolean              visible;       /*visibility*/
   Boolean              hilited;       /*highlighting*/
   Boolean              goAwayFlag;    /*presence of close box*/
   Boolean              spareFlag;     /*presence of zoom box*/
   RgnHandle            strucRgn;      /*handle to structure region*/
   RgnHandle            contRgn;       /*handle to content region*/
   RgnHandle            updateRgn;     /*handle to update region*/
   Handle               windowDefProc; /*handle to window definition */
                                       /* function*/
   Handle               dataHandle;    /*handle to window state data record*/
   StringHandle         titleHandle;   /*handle to window title*/
   short                titleWidth;    /*title width in pixels*/
   ControlHandle        controlList;   /*handle to control list*/
   struct CWindowRecord *nextWindow;   /*next window in window list*/
   PicHandle            windowPic;     /*handle to optional picture*/
   long                 refCon;        /*storage available to your */
                                       /* application*/
};

typedef struct CWindowRecord CWindowRecord;
typedef CWindowRecord *CWindowPeek;

struct WindowRecord {
   GrafPort             port;          /*window's graphics port*/
   short                windowKind;    /*class of the window*/
   Boolean              visible;       /*visibility*/
   Boolean              hilited;       /*highlighting*/
   Boolean              goAwayFlag;    /*presence of close box*/
   Boolean              spareFlag;     /*presence of zoom box*/
   RgnHandle            strucRgn;      /*handle to structure region*/
   RgnHandle            contRgn;       /*handle to content region*/
   RgnHandle            updateRgn;     /*handle to update region*/
   Handle               windowDefProc; /*handle to window definition */
                                       /* function*/
   Handle               dataHandle;    /*handle to window state data record*/
   StringHandle         titleHandle;   /*handle to window title*/
   short                titleWidth;    /*title width in pixels*/
   ControlHandle        controlList;   /*handle to window's control list*/
   struct WindowRecord  *nextWindow;   /*next window in window list*/
```

```
   PicHandle            windowPic;     /*handle to optional picture*/
   long                 refCon;        /*reference constant*/
};

typedef struct WindowRecord WindowRecord;
typedef WindowRecord *WindowPeek;

struct WStateData {
   Rect  userState;  /*user state*/
   Rect  stdState;   /*standard state*/
};

typedef struct WStateData WStateData;
typedef WStateData *WStateDataPtr, **WStateDataHandle;

struct AuxWinRec {
    struct AuxWinRec **awNext;      /*handle to next record*/
    WindowPtr        awOwner;       /*pointer to window */
    CTabHandle       awCTable;      /*handle to color table*/
    Handle           dialogCItem;   /*storage used by Dialog Manager*/
    long             awFlags;       /*reserved*/
    CTabHandle       awReserved;    /*reserved*/
    long             awRefCon;      /*reference constant, for use by */
                                    /* application*/
};

typedef struct AuxWinRec AuxWinRec;
typedef AuxWinRec *AuxWinPtr, **AuxWinHandle;

struct WinCTab {
   long        wCSeed;           /*reserved*/
   short       wCReserved;       /*reserved*/
   short       ctSize;           /*number of entries in table –1*/
   ColorSpec   ctTable[5];       /*array of color specification records*/
};

typedef struct WinCTab WinCTab;
typedef WinCTab *WCTabPtr, **WCTabHandle;
```

Window Manager Routines

Initializing the Window Manager

```
pascal void InitWindows(void);
```

Creating Windows

```
pascal WindowPtr GetNewCWindow
                            (short windowID, void *wStorage,
                             WindowPtr behind);
pascal WindowPtr GetNewWindow
                            (short windowID, void *wStorage,
                             WindowPtr behind);
pascal WindowPtr NewCWindow (void *wStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short procID, WindowPtr behind,
                             Boolean goAwayFlag, long refCon);
pascal WindowPtr NewWindow  (void *wStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short theProc, WindowPtr behind,
                             Boolean goAwayFlag, long refCon);
```

Naming Windows

```
pascal void SetWTitle       (WindowPtr theWindow, ConstStr255Param title);
pascal void GetWTitle       (WindowPtr theWindow, Str255 title);
```

Displaying Windows

```
pascal void DrawGrowIcon    (WindowPtr theWindow);
pascal void SelectWindow    (WindowPtr theWindow);
pascal void ShowWindow      (WindowPtr theWindow);
pascal void HideWindow      (WindowPtr theWindow);
pascal void ShowHide        (WindowPtr theWindow, Boolean showFlag);
pascal void HiliteWindow    (WindowPtr theWindow, Boolean fHilite);
pascal void BringToFront    (WindowPtr theWindow);
pascal void SendBehind      (WindowPtr theWindow, WindowPtr behindWindow);
```

Retrieving Mouse Information

```
pascal short FindWindow     (Point thePoint, WindowPtr *theWindow);
pascal WindowPtr FrontWindow(void);
```

Moving Windows

```
pascal void DragWindow      (WindowPtr theWindow, Point startPt,
                             const Rect *boundsRect);
pascal void MoveWindow      (WindowPtr theWindow, short hGlobal,
                             short vGlobal, Boolean front);
```

```
pascal long DragGrayRgn     (RgnHandle theRgn, Point startPt,
                             const Rect *boundsRect,
                             const Rect *slopRect,
                             short axis, DragGrayRgnProcPtr actionProc);
pascal long PinRect         (const Rect *theRect, Point *thePt);
```

Resizing Windows

```
pascal long GrowWindow      (WindowPtr theWindow, Point startPt,
                             const Rect *bBox);
pascal void SizeWindow      (WindowPtr theWindow, short w, short h,
                             Boolean fUpdate);
```

Zooming Windows

```
pascal Boolean TrackBox     (WindowPtr theWindow, Point thePt,
                             short partCode);
pascal void ZoomWindow      (WindowPtr theWindow, short partCode,
                             Boolean front);
```

Closing and Deallocating Windows

```
pascal Boolean TrackGoAway  (WindowPtr theWindow, Point thePt);
pascal void CloseWindow     (WindowPtr theWindow);
pascal void DisposeWindow   (WindowPtr theWindow);
```

Maintaining the Update Region

```
pascal void BeginUpdate     (WindowPtr theWindow);
pascal void EndUpdate       (WindowPtr theWindow);
pascal void InvalRect       (const Rect *badRect);
pascal void InvalRgn        (RgnHandle badRgn);
pascal void ValidRect       (const Rect *goodRect);
pascal void ValidRgn        (RgnHandle goodRgn);
```

Setting and Retrieving Other Window Characteristics

```
pascal void SetWindowPic    (WindowPtr theWindow, PicHandle pic);
pascal PicHandle GetWindowPic
                            (WindowPtr theWindow);
pascal void SetWRefCon      (WindowPtr theWindow, long data);
pascal long GetWRefCon      (WindowPtr theWindow);
pascal short GetWVariant    (WindowPtr theWindow);
```

Manipulating the Desktop

```
pascal void SetDeskCPat     (PixPatHandle deskPixPat);
#define GetGrayRgn()        (* (RgnHandle* 0X09EE))
pascal void GetCWMgrPort    (CGrafPtr *wMgrCPort);
pascal void GetWMgrPort     (GrafPtr *wPort);
```

Manipulating Window Color Information

```
pascal void SetWinColor     (WindowPtr theWindow,
                             WCTabHandle newColorTable);
pascal Boolean GetAuxWin    (WindowPtr theWindow, AuxWinHandle *awHndl);
```

Low-Level Routines

```
pascal Boolean CheckUpdate  (EventRecord *theEvent);
pascal void ClipAbove       (WindowPeek window;)
pascal void SaveOld         (WindowPeek window);
pascal void DrawNew         (WindowPeek window, Boolean update);
pascal void PaintOne        (WindowPeek window, RgnHandle clobberedRgn);
pascal void PaintBehind     (WindowPeek startWindow,
                             RgnHandle clobberedRgn);
pascal void CalcVis         (WindowPeek window);
pascal void CalcVisBehind   (WindowPeek startWindow,
                             RgnHandle clobberedRgn);
```

Application-Defined Routine

The Window Definition Function

```
pascal long MyWindow        (short varCode, WindowPtr theWindow,
                             short message, long param);
```

Assembly-Language Summary

Data Types

Window Record and Color Window Record Data Structure

0	windowPort	108 bytes	window's graphics port
108	windowKind	word	how window was created
110	wVisible	byte	visibility status
111	wHilited	byte	highlighted status
112	wGoAway	byte	presence of close box
113	wZoom	byte	presence of zoom box
114	structRgn	long	handle to structure region
118	contRgn	long	handle to content region
122	updateRgn	long	handle to update region
126	windowDef	long	handle to window definition function
130	wDataHandle	long	handle to window state data record
134	wTitleHandle	long	handle to window's title
138	wTitleWidth	word	title width in pixels
140	wControlList	long	handle to window's control list
144	nextWindow	long	pointer to next window in window list
148	windowPic	long	handle to picture for updates
152	wRefCon	long	reference constant field

Window State Data Structure

0	userState	8 bytes	user state rectangle
8	stdState	8 bytes	standard state rectangle

Window Color Information Table Data Structure

0	ctSeed	long	ID number for table
4	ctFlags	word	flags word
6	ctSize	word	number of entries minus 1
8	ctTable	variable	a series of color specification records (8 bytes each)

Auxiliary Window Record Data Structure

0	awNext	long	handle to next window in chain
4	awOwner	long	pointer to associated window record
8	awCTable	long	handle to window color information table
12	dialogCItem	long	handle to dialog color structures
16	awFlags	long	handle for QuickDraw
20	awResrv	long	reserved
24	awRefCon	long	user constant

Global Variables

AuxWinHead	Handle to beginning of auxiliary window list.
CurActivate	Pointer to window to receive activate event.
CurDeactive	Pointer to window to receive deactivate event.
DeskHook	Address of procedure for painting desktop.
DeskPattern	Pattern in which desktop is painted (8 bytes).
DragHook	Address of optional procedure to execute during TrackGoAway, TrackBox, DragWindow, GrowWindow, and DragGrayRgn.
DragPattern	Pattern of dragged region's outline (8 bytes).
GrayRgn	Handle to desktop region.
OldContent	Handle to saved content region.
OldStructure	Handle to saved structure region.
PaintWhite	Flag indicating whether to paint window white before update event (2 bytes).
SaveUpdate	Flag indicating whether to generate update events (2 bytes).
SaveVisRgn	Handle to saved visible region.
WindowList	Pointer to first window in window list.
WMgrPort	Pointer to Window Manager port.