Window Manager

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

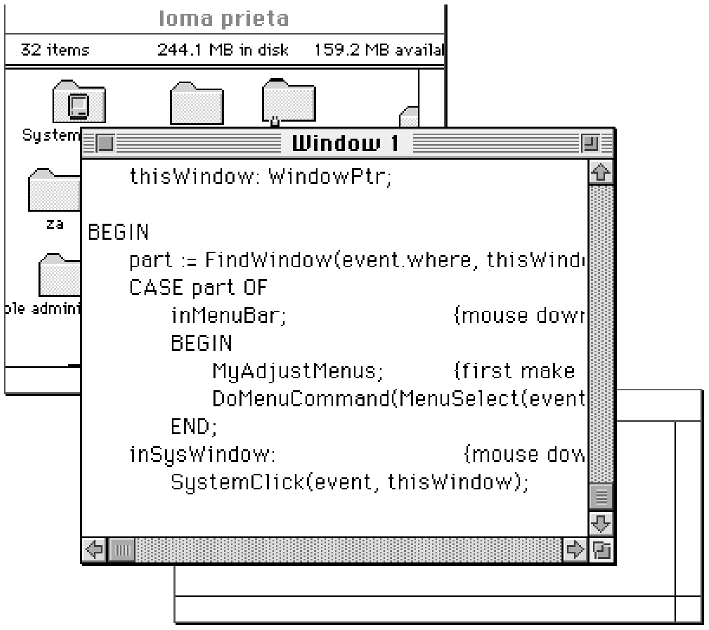
- create and display windows
- handle events in windows
- change the display when the user moves or resizes windows
- remove windows

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



```
    thisWindow: WindowPtr;

BEGIN
    part := FindWindow(event.where, thisWind
    CASE part OF
        inMenuBar;              {mouse dowr
        BEGIN
            MyAdjustMenus;      {first make
            DoMenuCommand(MenuSelect(event
        END;
    inSysWindow:                {mouse dow
        SystemClick(event, thisWindow);
```

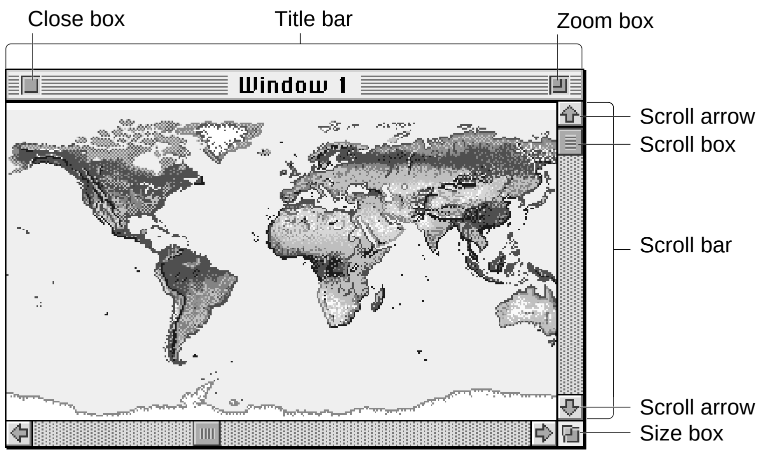
Your application typically creates **document windows** that allow the user to enter and display text, graphics, or other information. For an illustration of a document window in full color, see Plate 1 at the beginning of this book.

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.)

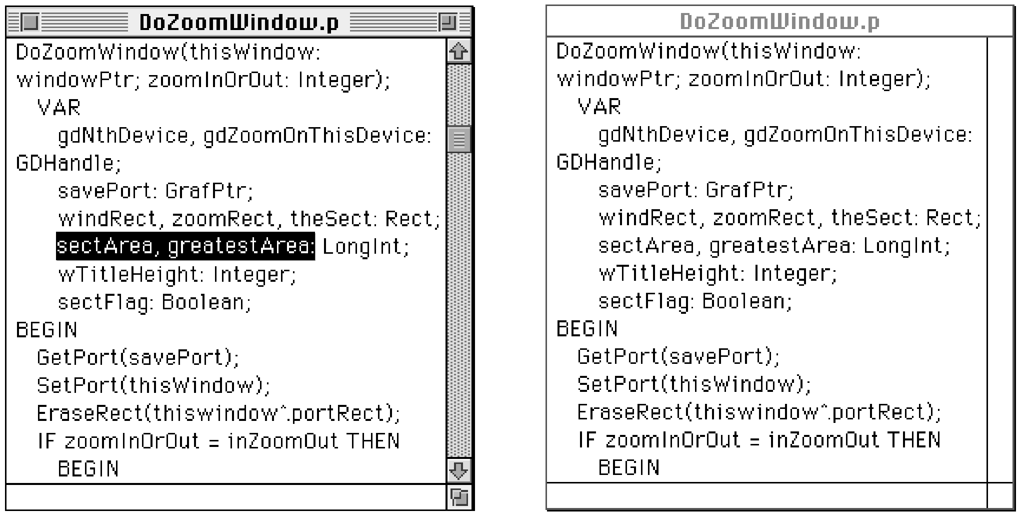
Active and Inactive Windows

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



```
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);
  IF zoomInOrOut = inZoomOut THEN
    BEGIN
```

Active document window

DoZoomWindow.p

```
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);
  IF zoomInOrOut = inZoomOut THEN
    BEGIN
```

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

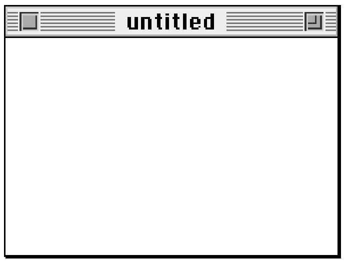
Because windows have so many uses, their appearances vary. The Window Manager defines a number of **window types** that meet the basic needs of most applications. A window type is the general description of how a window looks and behaves. Some windows have title bars and others don't, for example, and 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.

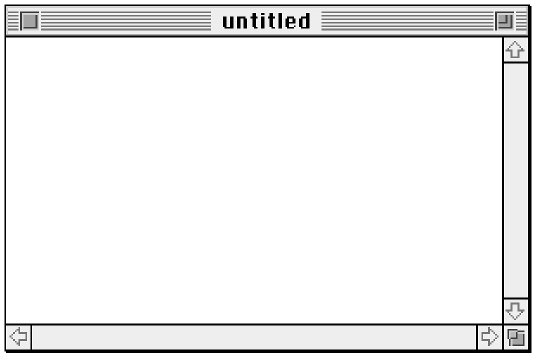
Figure 4-4 A window of type `zoomDocProc`



`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.

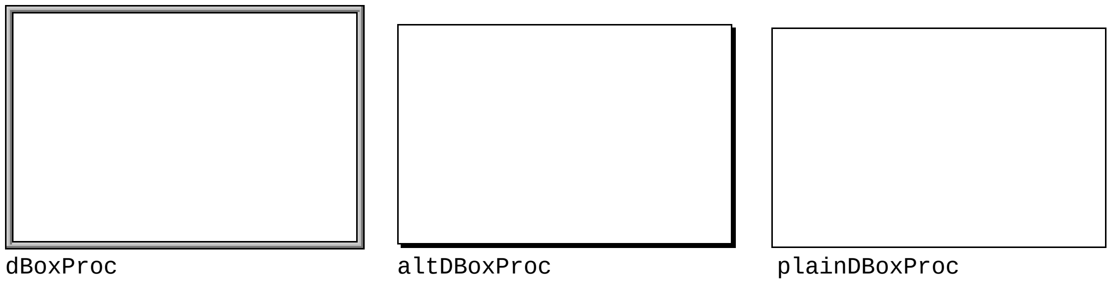
Figure 4-5 A window of type `zoomDocProc`, with size box and inactive scroll bars



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.

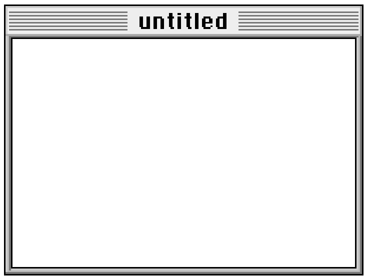
Alert boxes and fixed-position modal dialog boxes contain no window-manipulation elements. The user cannot move, resize, zoom, or close them manually. An alert box or a modal dialog box remains on the screen as the active window until the Dialog Manager or your application removes it—usually when the 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.

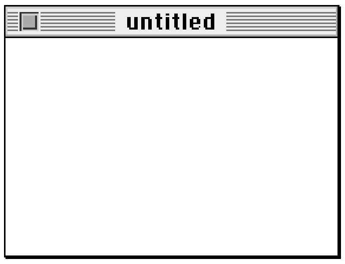
Figure 4-7 A window of type `movableDBoxProc`



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`

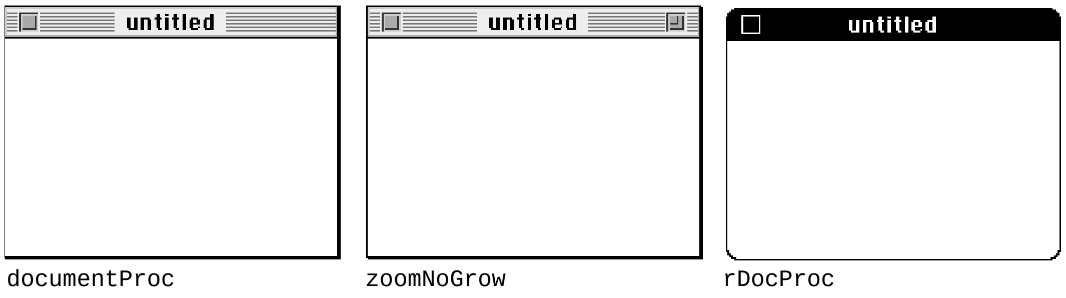


noGrowDocProc

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.

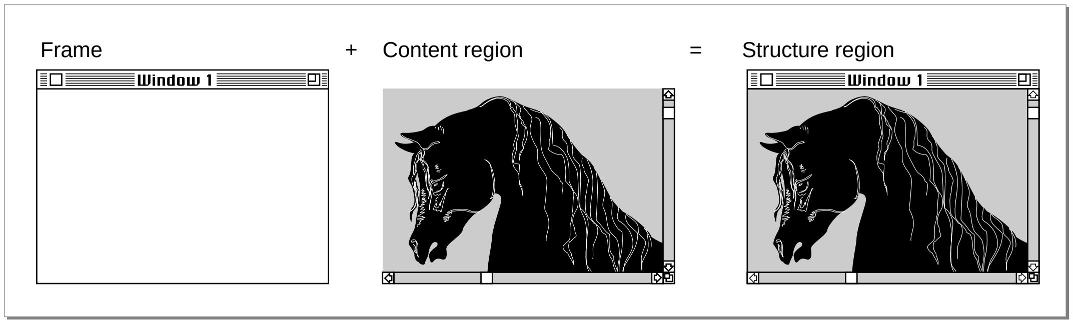
The most obvious window regions are the parts of the visible window that the user manipulates to control the display. These window regions correspond to the standard window parts. The **drag region** is the area occupied by the title bar, except for the close box and zoom box. (The user moves the window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

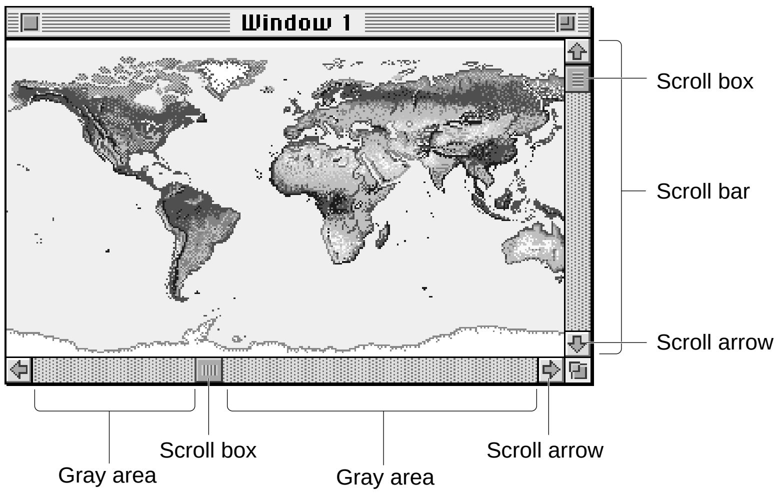
Use the `noGrowDocProc` window type for modeless dialog boxes. You typically use the Dialog Manager to handle events in a modeless dialog box, much like events in a movable modal dialog box. Your application handles window-manipulation events in modeless dialog boxes just as it handles them in document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

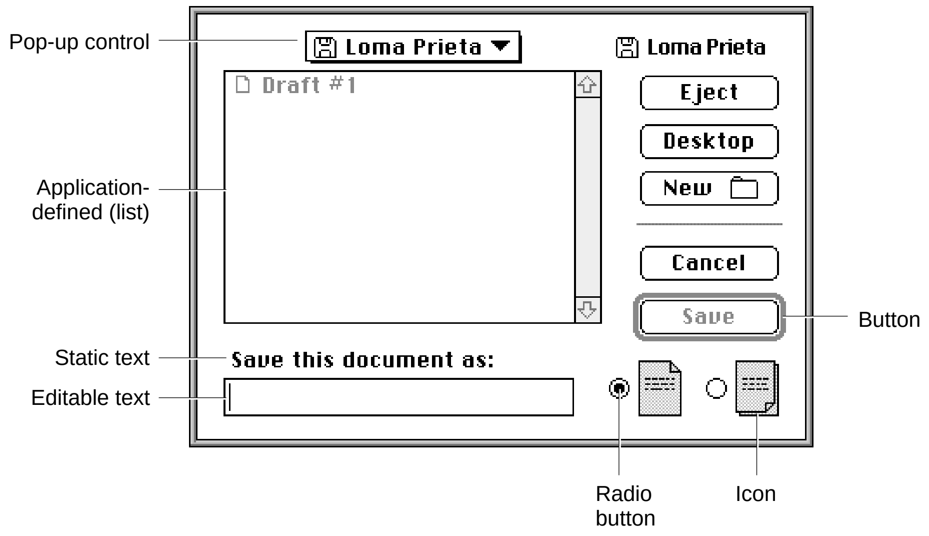


You use scroll bars to show the relative position, within the entire document, of the portion of the document displayed in the window. You should allow the user to drag the scroll box or click in the gray areas or the scroll arrows to move parts of the document into and out of the window. You activate 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.

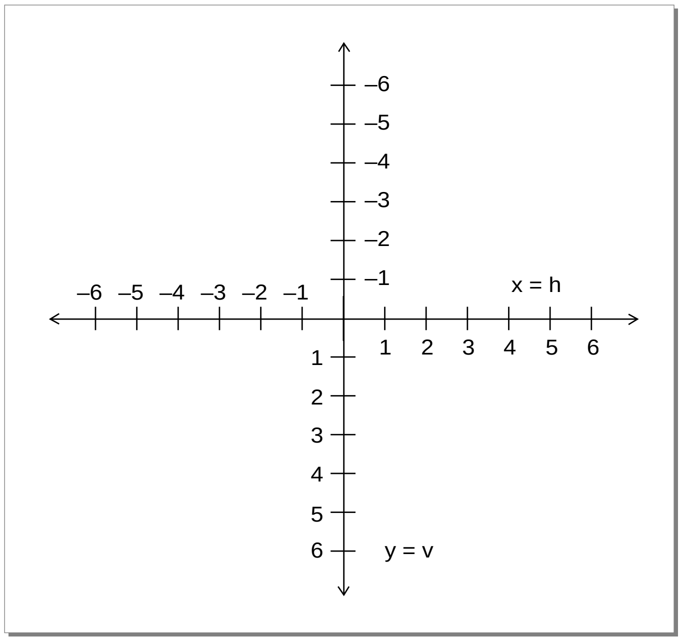
The Window Manager has built-in support for the nine basic window types described in "Types of Windows" beginning on page 4-8. When you are using one of these window types, the Window Manager draws the window's frame, determines what region of the window the cursor is in, calculates the window's structure 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.

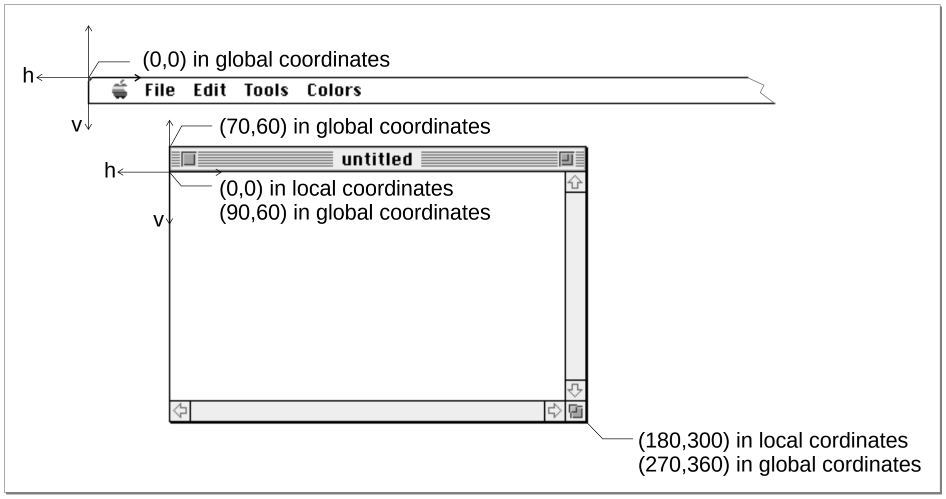
Note
When you place a window on the screen, you specify the location of its content region, in global coordinates. Remember to allow space for the window's title bar. On the main screen, remember to leave space for the menu bar. In the Roman script system, both the standard document title bar and the 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


(0,0) in global coordinates
h
v
File Edit Tools Colors
(70,60) in global coordinates
untitled
h
v
(0,0) in local coordinates
(90,60) in global coordinates
(180,300) in local cordinates
(270,360) in global cordinates

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.

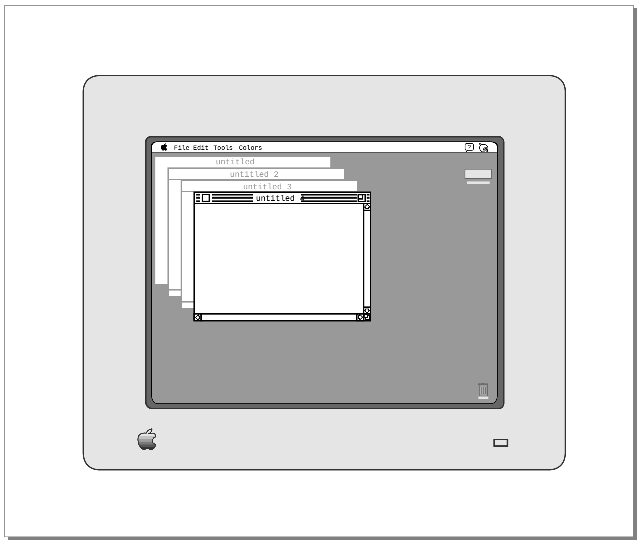
- If you're creating a window because the user is creating a new document, you can display the window immediately by calling the procedure `ShowWindow` to make the window frame visible. This change in visibility adds to the update region and triggers an update event. Your application then invokes its 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.

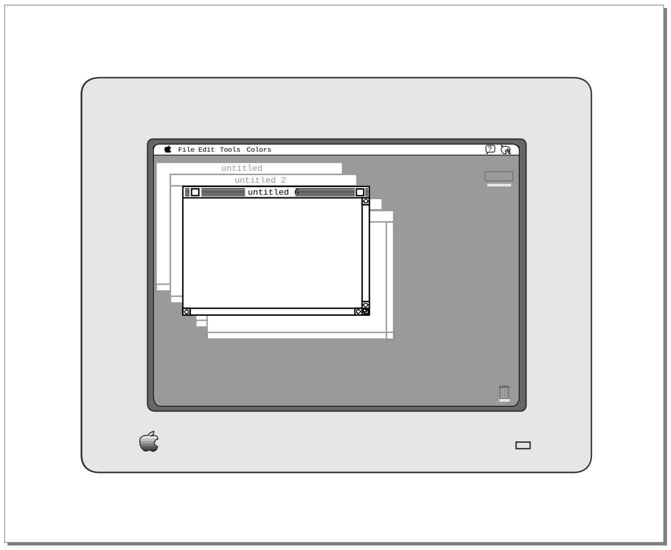
On Macintosh computers with a single screen of known size, positioning windows is fairly straightforward. You position the first new document window on the upper-left corner of the desktop. Open each additional new document window with its upper-left corner slightly below and to the right of the upper-left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

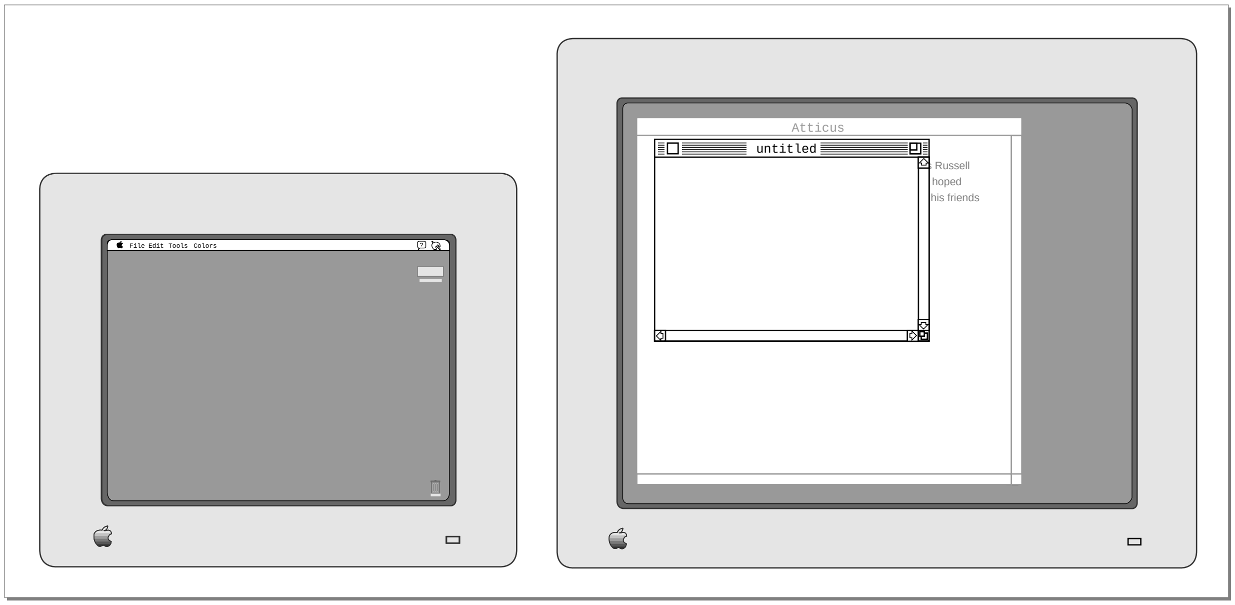
If the user moves or closes a window that occupies one of the interim positions, and the window template specifies `staggerParentWindowScreen`, the Window Manager uses the "empty" slot for the next new window created before moving further down and to the right.

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.

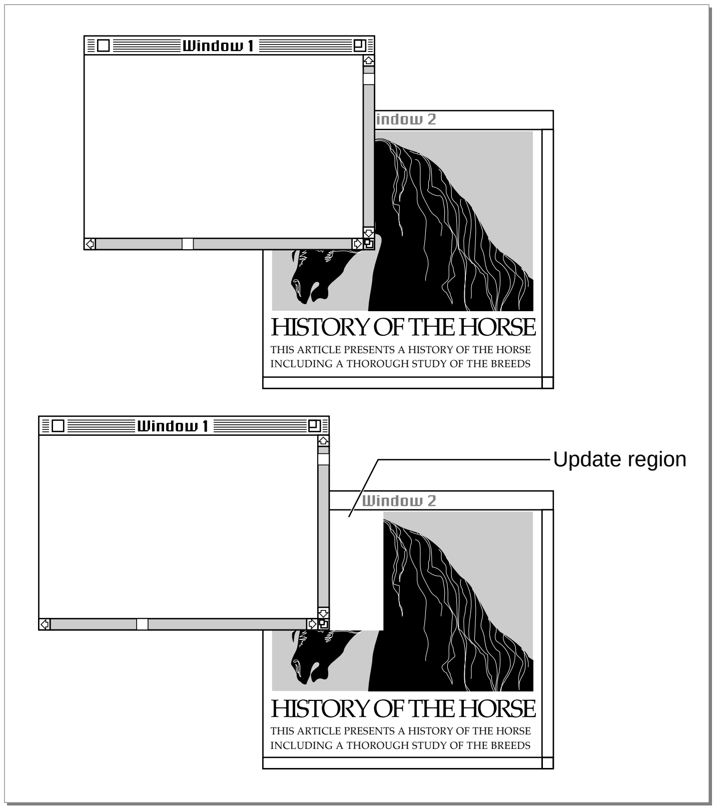
If your application creates a window that contains a static display, you can let the Window Manager take care of drawing and updating the content region by placing a handle to a picture in the `windowPic` field of the window record. See the description of the `SetWindowPic` procedure on page 4-110.

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


Window 1
Window 2
HISTORY OF THE HORSE
THIS ARTICLE PRESENTS A HISTORY OF THE HORSE INCLUDING A THOROUGH STUDY OF THE BREEDS
Window 1
Update region
Window 2
HISTORY OF THE HORSE
THIS ARTICLE PRESENTS A HISTORY OF THE HORSE INCLUDING A THOROUGH STUDY OF THE BREEDS

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.

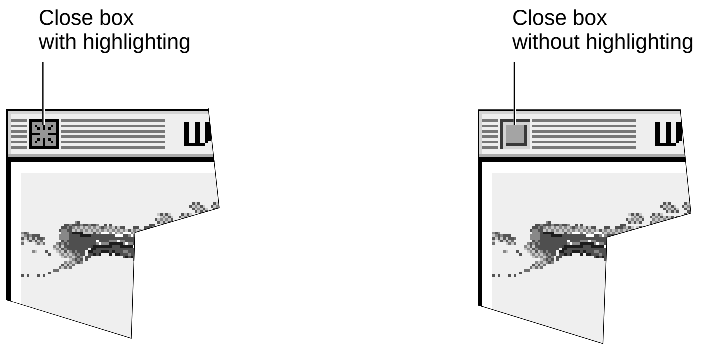
If the cursor is in the content area of the active window, the DoMouseDown procedure calls another application-defined procedure (DoContentClick) that handles mouse events in the content area.

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

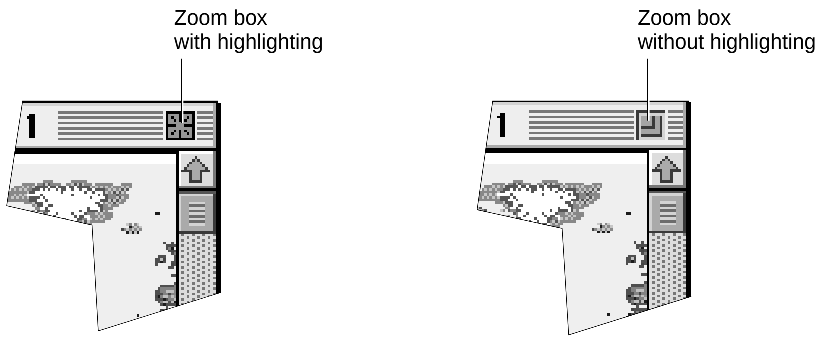


When the user releases the mouse button, `TrackGoAway` returns `TRUE` if the cursor is still in the close box and `FALSE` if it is not. If `TrackGoAway` returns `TRUE`, `DoMouseDown` calls the application-defined procedure `DoCloseCmd` to close the window. Listing 4-16 on page 4-60 shows the `DoCloseCmd`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

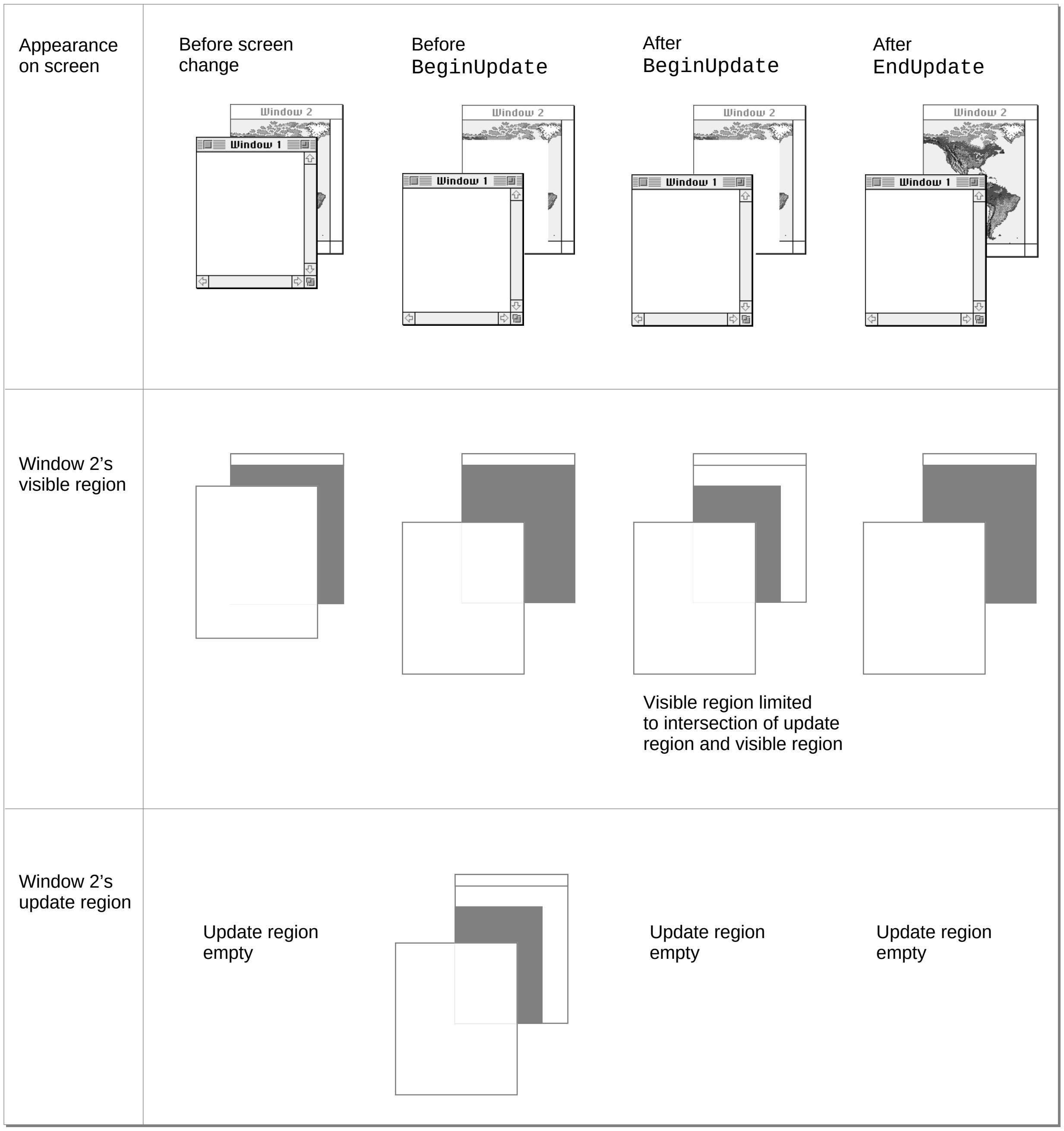
Your application can receive update events when it is in either the foreground or the background.

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.

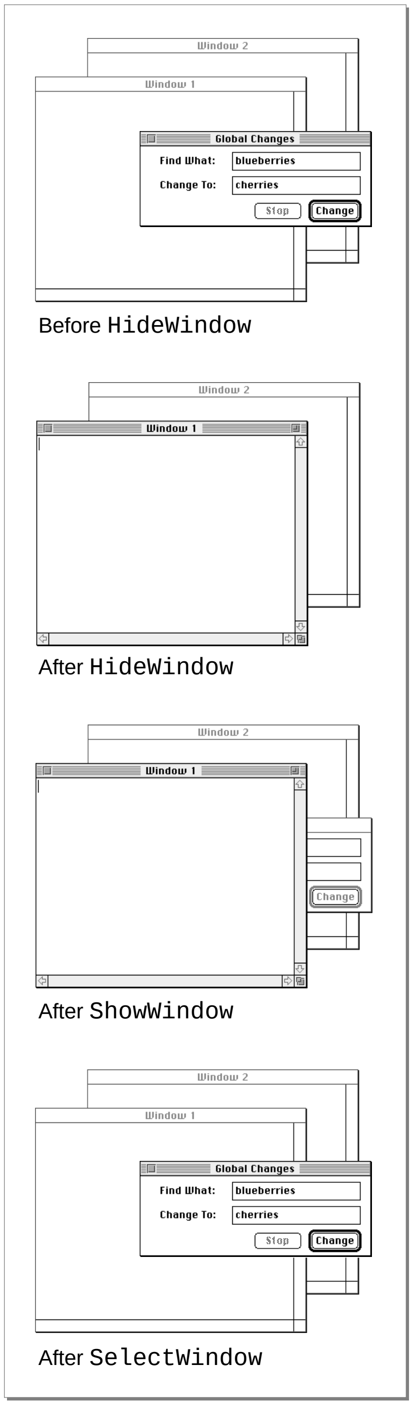
Hiding and Showing a Window

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.