Control Manager

This chapter describes how your application can use the Control Manager to create and manage controls. **Controls** are onscreen objects that the user can manipulate with the mouse. By manipulating controls, the user can take an immediate action or change settings to modify a future action. For example, a scroll bar control allows a user to immediately change the portion of the document that your application displays, whereas a pop-up menu control for baud rate might allow the user to change the rate by which your application handles subsequent data transmissions.

Read this chapter to learn how and when to implement controls. Virtually all applications need to implement controls, at least in the form of scroll bars for document windows. You use Control Manager routines, resources, and data structures to implement scroll bars in your application's document windows.

The other standard Macintosh controls are buttons, checkboxes, radio buttons, and pop-up menus. You can use the Control Manager to create and manage these controls, too. Alternatively, you can use the Dialog Manager to implement these controls in alert boxes and dialog boxes more easily. (You typically use an alert box to warn a user of an unusual situation, and you typically use a dialog box to ask the user for information necessary to carry out a command.) The chapter "Dialog Manager" in this book describes in detail how to implement controls in alert and dialog boxes. However, in certain situations—for instance, when you need to implement highly complex dialog boxes—you may want to use Control Manager routines to manage these types of controls directly; read this chapter for information on how to do so.

For scrolling lists of graphic or textual information (similar to the list of files that system software presents after the user chooses the Open command from the File menu), your application can use the List Manager to implement the scroll bars. See the chapter "List Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information.

The Control Manager offers routines for automatically handling user-generated mouse events in controls and redrawing controls in response to update events. For further information about events and event handling, see the chapter "Event Manager" in this book.

You typically use a control resource—a resource of type `'CNTL'`—to specify the type, size, location, and other attributes of a control. See the chapter "Introduction to the Macintosh Toolbox" in this book for general information about resources; detailed information about the Resource Manager and its routines is provided in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox.*

Every control you create must be associated with a particular window. All of the controls for a window are stored in a control list referenced by the window's window record. See the chapter "Window Manager" in this book for general information about windows. (When you use the Dialog Manager to implement a control, the Dialog Manager associates it with its respective dialog box or alert box, as described in the chapter "Dialog Manager.")

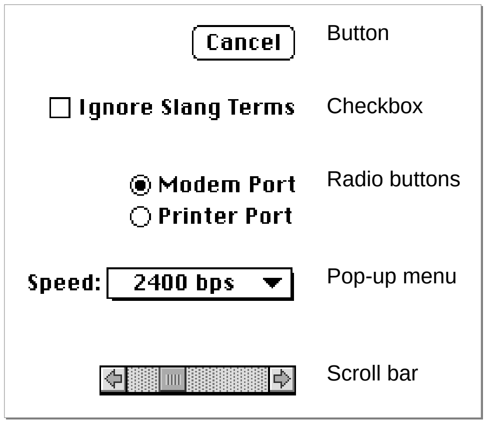
This chapter provides an introduction to the use of controls, and then discusses how you can

- create and display controls
- determine whether mouse-down events have occurred in controls
- respond to mouse-down events in controls
- change the settings in controls
- use scroll bars to move a document in a window
- move and resize controls for a window
- define your own control definition function to create nonstandard controls

Introduction to Controls

The Control Manager provides several standard controls. Figure 5-1 illustrates these standard controls: buttons, checkboxes, radio buttons, pop-up menus, and scroll bars. You can also design and implement your own custom controls.

Figure 5-1 Standard controls provided by the Control Manager



Buttons, checkboxes, and radio buttons are the simplest controls. They consist of only a title and an outline shape, and they respond to only mouse clicks. A pop-up menu is slightly more complex. This control has a menu attached to its title, and it must respond when the user drags the cursor across the menu. A scroll bar, because it consists of different parts that behave differently, is the most complex of the standard controls. Even though a scroll bar has several parts, it is still only one control.

The Control Manager displays these standard controls in colors that provide aesthetic consistency across all monitors, from black-and-white displays to 8-bit color displays. To ensure consistency across applications, you generally shouldn't change the default colors of controls, although the Control Manager does allow you to do so with the `SetControlColor` procedure (described on page 5-101) or the control color table resource (described on page 5-121).

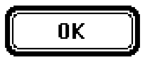
Standard controls and common custom controls are described in the next several sections.

Buttons

Buttons appear on the screen as rounded rectangles with a title centered inside. When the user clicks a button, your application should perform the action described by the button title. Typically, buttons allow the user to perform actions instantaneously—for example, completing the operations defined by a dialog box or acknowledging an error message in an alert box.

Make your buttons large enough to surround their titles. In every window or dialog box in which you display buttons, you should designate one button as the default button by drawing a thick black outline around it (as shown in Figure 5-2). Your application should respond to key-down events involving the Enter and Return keys as if the user had clicked the default button. (In your alert boxes, the Dialog Manager automatically outlines the default button; you must outline the default button in your dialog boxes.)

Figure 5-2 A default button

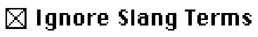


You normally use buttons in alert boxes and dialog boxes. See the chapter "Dialog Manager" for additional details about where to display buttons, what to title them, how to respond to events involving them, and how to draw an outline around them.

Checkboxes

Checkboxes provide alternative choices. Typically you use checkboxes in dialog boxes so that users can specify information necessary for completing a command. Checkboxes act like toggle switches, turning a setting either off or on. Use checkboxes to indicate one or more options that must be either off or on. A checkbox appears as a small square with a title alongside it; use the Control Manager procedure `SetControlValue` to place an X in the box when the user selects it by clicking it on and to remove the X when the user deselects it by clicking it off. Figure 5-3 shows a selected checkbox.

Figure 5-3 A selected checkbox



When you design a dialog box, you can include any number of checkboxes—including only one. Checkboxes are independent of each other, even when they offer related options. Within a dialog box, it's a good idea to group sets of related checkboxes and to provide some visual demarcation between different groups.

Each checkbox has a title. It can be very difficult to title the option in an unambiguous way. The title should reflect two clearly opposite states. For example, in a Finder's Info window, a checkbox provides the option to lock a file. The checkbox is titled simply Locked. The clearly opposite state, when the option is off, is unlocked.

If you can't devise a checkbox title that clearly implies an opposite state, you might be better off using two radio buttons. With two radio buttons, you can use two titles, thereby clarifying the states.

Checkboxes are frequently used in dialog boxes to set or modify future actions instead of specifying actions to be taken immediately. See the chapter "Dialog Manager" in this book for a detailed discussion of how and where to display checkboxes in dialog boxes.

Radio Buttons

Like checkboxes, **radio buttons** retain and display an on-or-off setting. You organize radio buttons in a group to offer a choice among several alternatives—typically, inside a dialog box. Radio buttons are small circles; when the user clicks a radio button to turn it on, use the Control Manager procedure `SetControlValue` to fill the radio button with a small black dot. The user can have only one radio button setting in effect at one time. In other words, radio buttons are mutually exclusive. However, the Control Manager cannot determine how your radio buttons are grouped; therefore, when the user turns on one radio button, it is up to your application to use `SetControlValue` to turn off the others in that group.

A set of radio buttons normally has two to seven items; each set must always have at least two radio buttons. Each set of radio buttons must have a label that identifies the kind of choices the group offers. Also, each button must have a title that identifies what the radio button does. This title can be a few words or a phrase. A set of radio buttons is *never* dynamic—that is, its contents should never change according to the context. (If you need to display more than seven items, or if the items change as the context changes, you should use a pop-up menu instead.)

Radio buttons represent choices that are related but not necessarily opposite. For example, a pair of radio buttons may provide a choice between using the modem port or the printer port, as shown in Figure 5-1 on page 5-4. If more than one set of radio buttons is visible at one time, you need to demarcate the sets from one another. For example, you can draw a dotted line around a set of radio buttons to separate it from other elements in a dialog box.

Pop-Up Menus

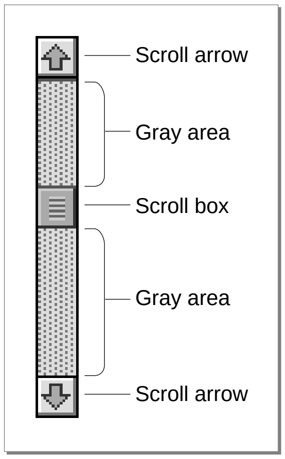
Pop-up menus, introduced in the chapter "Menu Manager" in this book, provide the user with a simple way to choose from among a list of choices without having to move the cursor to the menu bar. As an alternative to a group of radio buttons, a pop-up menu is particularly useful for specifying a group of settings or values that number five or more, or whose settings or values might change. Like the items in a set of radio buttons, the items in a pop-up menu are mutually exclusive—that is, only one choice from the menu can be in effect at any time. Figure 5-8 on page 5-12 illustrates the choices available in a pop-up menu that has been selected by the user.

Never use a pop-up menu as a way to provide the user with commands. Pop-up menus should not list actions (that is, verbs); instead, they should list attributes (that is, adjectives) or settings from which the user can choose one option.

Scroll Bars

Scroll bars change what portion of a document the user can view within the document's window. A scroll bar is a light gray rectangle with scroll arrows at each end. Inside the scroll bar is a square called the **scroll box.** The rest of the scroll bar is called the **gray area.** Windows can have a horizontal scroll bar, a vertical scroll bar, or both. A vertical scroll bar lies along the right side of a window. A horizontal scroll bar runs along the bottom of a window. Figure 5-4 shows the parts of a scroll bar.

Figure 5-4 A vertical scroll bar



If the user drags the scroll box, clicks a scroll arrow, or clicks anywhere in the gray area, your application "moves" the document accordingly; use Control Manager routines as appropriate to move the scroll box. Figure 5-5 illustrates, and the next few sections explain, several key behaviors of a scroll bar.

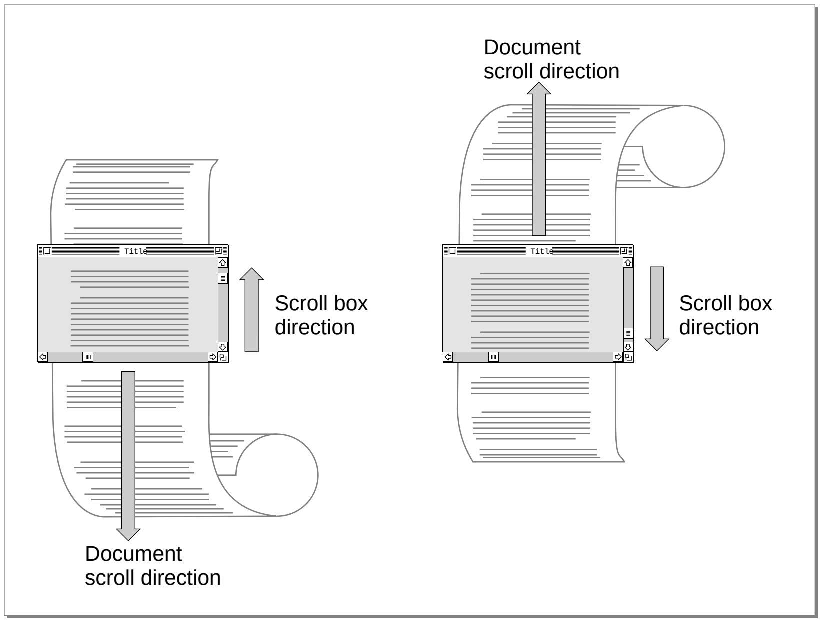
A scroll bar represents the entire document in one dimension, top to bottom or right to left. The scroll box shows the position, relative to the whole document, of the visible portion of the document. If the scroll box is halfway between the top and bottom of the scroll bar, then what the user sees should be about halfway through the document. Use the `SetControlValue` or `SetControlMaximum` procedure to move the scroll box whenever your application resizes a window and whenever it scrolls through a document for any reason other than responding to the user dragging the scroll box.

After the user drags the scroll box, the Control Manager redraws the scroll box in its new position. You then use the `GetControlValue` function to determine the position of the scroll box, and you display the appropriate portion of the document. By dragging the scroll box, the user can move quickly through the document. For example, to see the beginning of the document, the user drags the scroll box to the top of the scroll bar. Your application then scrolls to the top of the document.

At either end of the scroll bar are **scroll arrows** that indicate the direction of movement through the document. For instance, when the user clicks the top scroll arrow, your application needs to move toward the beginning of the document. Thus, the document moves down, seemingly in the opposite direction. By clicking the scroll arrow, the user tells your application, “Show me more of the document that’s hidden in this direction.”

Your application uses the `SetControlValue` procedure to move the scroll box in the direction of the arrow being clicked. In this way, the scroll box continues to represent the approximate position of the visible part of the document in relation to the whole document. For example, when the user clicks the top scroll arrow, you move the document down to bring more of the top of the document into view, and you move the scroll box up, as illustrated in Figure 5-5.

Figure 5-5 Using the scroll box and scroll arrows



Each click of a scroll arrow should move the document a distance of one unit in the chosen direction. Your application determines what one unit equals. For example, a word processor should move one line of text for each click in the arrow. A spreadsheet should move one row or one column, depending on the direction of the arrow. To ensure smooth scrolling effects, it's usually best to specify the same size units within a document. When the user holds down the mouse button while the cursor is in a scroll arrow, your application should continuously scroll through the document in the indicated direction until the user releases the mouse button or your application has scrolled as far as possible.

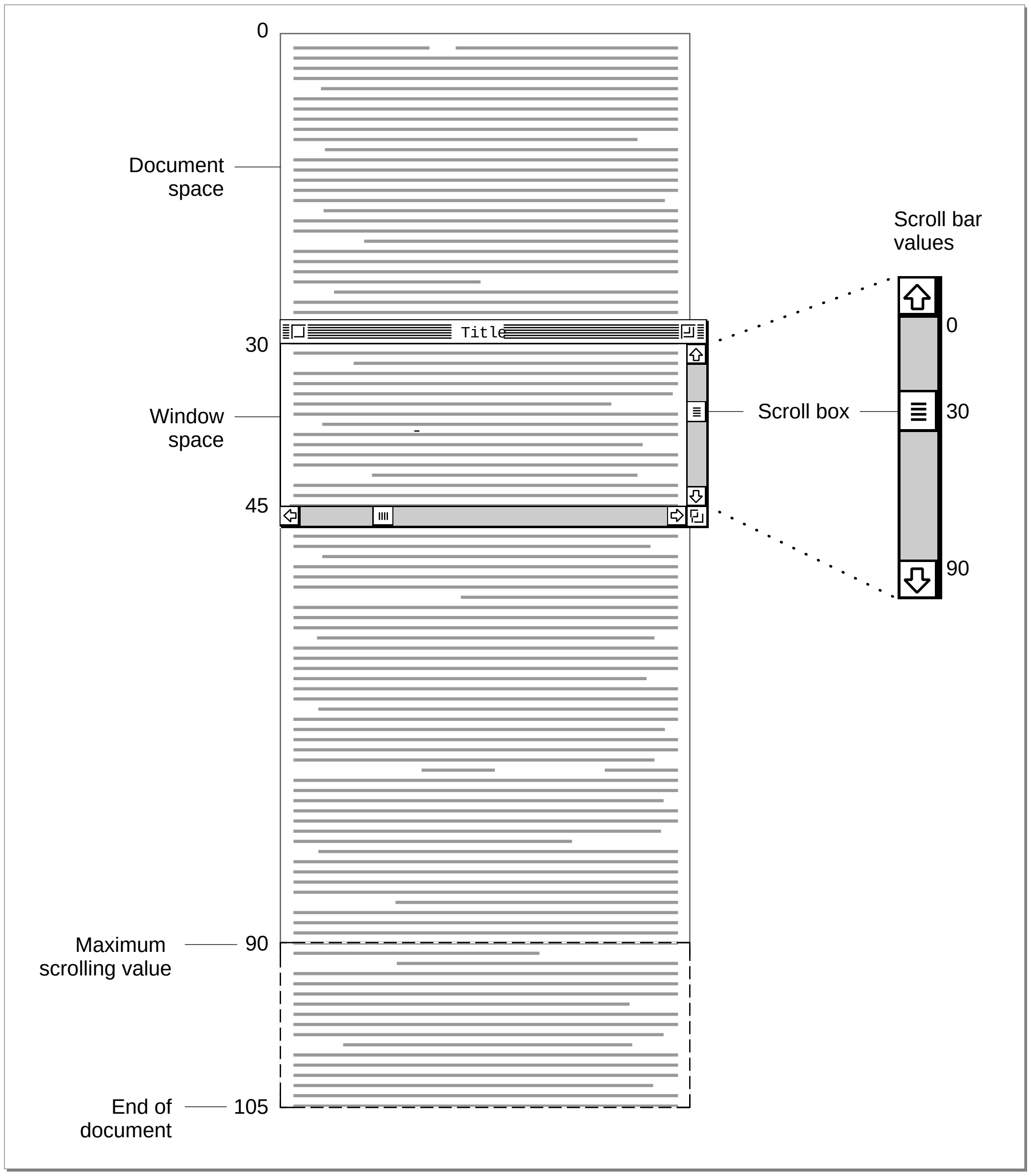
The rest of the area within the scroll bar—excluding the scroll box and the scroll arrows—is called the gray area. When the user clicks the gray area of a scroll bar, your application should move the displayed area of the document by an entire window of information minus one scroll unit. For example, if the window displays 15 lines of text and the user clicks the gray area below the scroll box, your application should move the document up 14 lines so that the bottom line of the previous view appears at the top of the new view. (This retained line helps the user see the newly displayed material in context.) You must also move the scroll box an appropriate distance in that direction. For example, when the user clicks the gray area below the scroll box, move the document view by one window toward the bottom of the document and use `SetControlValue` to move the scroll box accordingly.

When your application scrolls through a document—for example, when the user manipulates a scroll bar—your application must move the document's coordinate space in relation to the window's coordinate space. Your application uses the scroll box to indicate the location of the top of the displayed portion of the document relative to the rest of the document.

For example, if a text window contains 15 lines of text and the user scrolls 30 lines from the top of the document, the scroll box should be set to a value of 30. The window displays all of the lines between line 30 and line 45, as shown in Figure 5-6 on the next page. The scroll box always indicates the displacement between the beginning of the document and the top of the displayed portion of the document.

To prevent the user from scrolling past the edge of the document and seeing a blank window, you should—for a vertical scroll bar—allow the document to scroll no farther than the length of the document minus the height of the window, excluding the 15-pixel-deep region for the horizontal scroll bar at the bottom edge of the window. Likewise, for a horizontal scroll bar, you should allow the document to scroll no farther than the width of the document minus the width of the window—here, too, excluding the 15-pixel-wide region for the vertical scroll bar at the right edge of the window.

Figure 5-6 Spatial relations between a document and a window, and their representation by a scroll bar



For example, the document shown in Figure 5-6 is 105 lines long. So that the last 15 lines will fill the window when the user scrolls to the end of the document, the application does not scroll beyond 90 lines. Because the user has scrolled to line 30 of a maximum 90 lines, the scroll box appears a third of the way down the scroll bar.

“Scrolling Through a Document” beginning on page 5-43 describes in detail how to scroll through a document in a window.

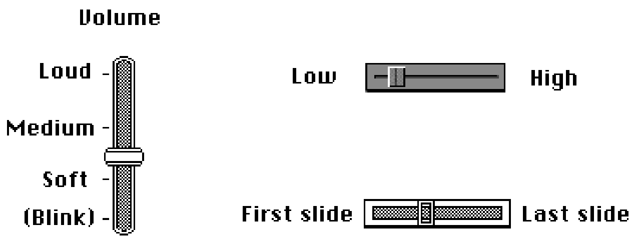
Other Controls

If you need controls other than the standard ones provided by the Control Manager, you can design and implement your own. Typically, the only types of controls you might need to implement are sliders or dials. **Sliders** and **dials** (which differ only in appearance) are similar to scroll bars in that they graphically represent a range of values that a user can set. Use an **indicator**—such as a sliding switch or a dial needle—to indicate the current setting for the control and to let the user set its value. (For scroll bars, the scroll box is the indicator.)

If you want to display a value not under the user's direct control (for example, the amount of free space remaining on a disk), you should use a status bar or other type of graphic instead of a slider or dial.

Figure 5-7 illustrates several custom controls, which are used for purposes such as setting the speaker volume, the gray-scale saturation level, and the relative position of a slide within a presentation. As in this figure, be sure to include meaningful labels that indicate the range and the direction of your control's indicator.

Figure 5-7 Custom slider controls



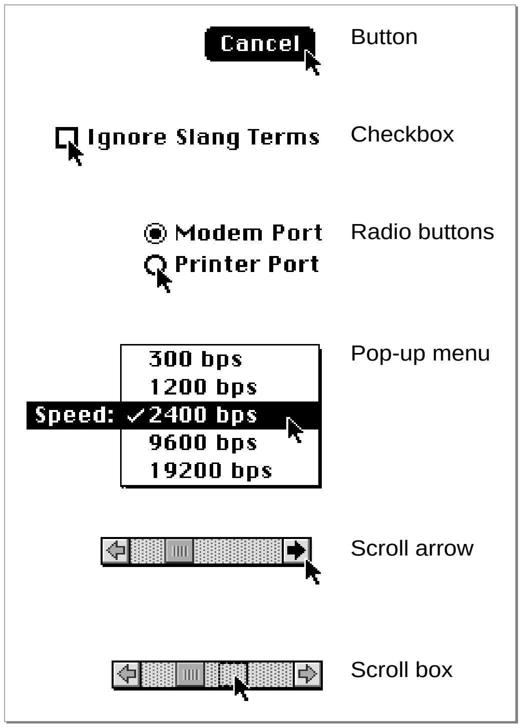
A scroll bar is a slider representing the entire contents of a window, and the user uses the scroll box to move to a specific location in that content. Don't use scroll bars to represent any other concept (for instance, changing a setting). Otherwise, your departure from the consistent Macintosh interface might confuse the user.

Active and Inactive Controls

You can make a control become either active or inactive. Figure 5-8 on the next page shows how the `TrackControl` function (which you use in response to a mouse-down event in a control) gives visual feedback when the user moves the cursor to an **active control** and presses the mouse button. In particular, `TrackControl` responds to mouse-down events in active controls by

- displaying buttons in inverse video
- drawing checkboxes and radio buttons with heavier lines
- highlighting the titles of and displaying the items in pop-up menus
- highlighting scroll arrows
- moving outlines of scroll boxes when users drag them

Figure 5-8 Visual feedback for user selection of active controls



Your application, in turn, should respond appropriately to mouse events involving active controls. Most often, your application waits until the user releases the mouse button before taking any action; as long as the user holds down the mouse button when the cursor is over a control, you typically let `TrackControl` react to the mouse-down event; `TrackControl` then informs your application the moment the user releases the mouse button when the cursor is over an active control.

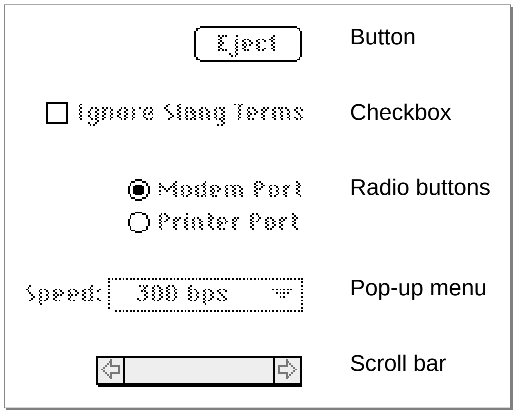
As soon as the user releases the mouse button, your application should

- perform the task identified by the button title when the cursor is over an active button
- toggle the value of the checkbox when the cursor is over an active checkbox (The Control Manager then draws or removes the checkmark, as appropriate.)
- turn on the radio button and turn off all other radio buttons in the group when the cursor is over an active radio button
- use the new setting chosen by the user when the cursor is over an active pop-up menu
- show more of the document in the direction of the scroll arrow when the cursor is over the scroll arrow or gray area of an active scroll bar, and move the scroll box accordingly
- determine where the user has dragged the scroll box when the cursor is over the scroll box and then display the corresponding portion of the document

Sometimes your application should respond even before the user releases the mouse button—that is, your application should undertake some continuous action as long as the user holds down the mouse button when the cursor is in an active control. Most typically, when the user moves the cursor to a scroll arrow or gray area and then holds down the mouse button, your application should continuously scroll through the document until the user releases the mouse button or until the user can't scroll any farther. To perform this kind of action, you define an **action procedure** and specify it to `TrackControl`; `TrackControl` calls your action procedure as long as the user holds down the mouse button.

Whenever it is inappropriate for your application to a respond to a mouse-down event in a control, you should make it inactive. An **inactive control** is one that the user can't use because it has no meaning or effect in the current context—for example, the scroll bars in an empty window. The Control Manager continues to display an inactive control so that it remains visible, but in a manner that indicates its state to the user. As shown in Figure 5-9, the Control Manager dims inactive buttons, checkboxes, radio buttons, and pop-up menus, and it lightens the gray area and removes the scroll box from inactive scroll bars.

Figure 5-9 Inactive controls



You can use the `HiliteControl` procedure to make any control inactive and then active again. Except for scroll bars (which you should hide using the `HideControl` procedure), you should use `HiliteControl` to make all other controls inactive when their windows are not frontmost. You typically use controls other than scroll bars in dialog boxes. See the chapter "Dialog Manager" in this book for a discussion of how to make buttons, radio buttons, checkboxes, and pop-up menus inactive and active.

You make scroll bars inactive when the document is smaller than the window in which you display it. To make a scroll bar inactive, you typically use the `SetControlMaximum` procedure to make the scroll bar's maximum value equal to its minimum value, in which case the Control Manager automatically makes the scroll bar inactive. To make it active again, you typically use `SetControlMaximum` to make its maximum value larger than its minimum value.

The Control Definition Function

A **control definition function** determines how a control generally looks and behaves. Various Control Manager routines call a control definition function whenever they need to perform some control-dependent action, such as drawing the control on the screen.

Control definition functions are stored as resources of type `'CDEF'`. The System file includes three **standard control definition functions,** stored with resource IDs of 0, 1, and 63. The `'CDEF'` resource with resource ID 0 defines the look and behavior of buttons, checkboxes, and radio buttons; the `'CDEF'` resource with resource ID 1 defines the look and behavior of scroll bars; and the `'CDEF'` resource with resource ID 63 defines the look and behavior of pop-up menus. (If you want to define nonstandard controls, you'll have to write control definition functions for them, as described in "Defining Your Own Control Definition Function" beginning on page 5-109.)

Just as a window definition function can describe variations of the same basic window, a control definition function can use a **variation code** to describe variations of the same basic control. You specify a particular control with a control definition ID. The **control definition ID** is an integer that contains the resource ID of the control definition function in its upper 12 bits and a variation code in its lower 4 bits. For a given resource ID and variation code, the control definition ID is derived as follows:

control definition ID = 16 * (`'CDEF'` resource ID) + variation code

For example, buttons, checkboxes, and radio buttons all use the standard control definition function with resource ID 0; because they have variation codes of 0, 1, and 2, respectively, their respective control definition IDs are 0, 1, and 2.

You can use these constants to define the controls provided by the standard control definition functions:

Constant	Control definition ID	Control
`pushButProc`	0	Button
`checkBoxProc`	1	Checkbox
`radioButProc`	2	Radio button
`scrollBarProc`	16	Scroll bar
`popupMenuProc`	1008	Pop-up menu

The control definition function for scroll bars figures out whether a scroll bar is vertical or horizontal from a rectangle you specify when you create the control.

About the Control Manager

You can use the Control Manager to

- create and dispose of controls
- display, update, and hide controls

- change the size, location, and appearance of controls
- monitor and respond to the user's operation of a control
- determine and change the settings and other attributes of a control

Your application performs these actions by calling the appropriate Control Manager routines. The Control Manager carries out the actual operations, but it's up to you to decide when, where, and how to carry these out.

Using the Control Manager

To implement a control, you generally

- use a control resource (that is, a resource of type `'CNTL'`) to describe the control
- create and display the control
- determine when the user presses, clicks, or holds down the mouse button while the cursor is in the control
- respond as appropriate to events involving the control—for example, by displaying a different portion of the document when the user manipulates a scroll bar
- respond as appropriate to other events in windows that include controls—for example, by moving and resizing a scroll bar when the user resizes a window, or by hiding one window's scroll bars when the user makes a different window active

These tasks are explained in greater detail in the rest of this chapter.

Before using the Control Manager, you must initialize QuickDraw, the Font Manager, and the Window Manager, in that order, by using the `InitGraf`, `InitFonts`, and `InitWindows` procedures. (See *Inside Macintosh: Imaging* for information about `InitGraf` and `InitFonts`; see the chapter "Window Manager" in this book for information about `InitWindows`.)

Creating and Displaying a Control

To create a control in one of your application's windows, use the `GetNewControl` or `NewControl` function. You should usually use `GetNewControl`, which takes information about the control from a control resource (that is, a `'CNTL'` resource) in a resource file. Like window resources, control resources isolate descriptive information from your application code for ease of modification—especially for translation to other languages. The rest of this section describes how to use `GetNewControl`. Although it's generally not recommended, you can also use the `NewControl` function and pass it the necessary descriptive information in individual parameters instead of using a control resource. The `NewControl` function is described on page 5-82.

When you use `GetNewControl`, you pass it the resource ID of the control resource, and you pass it a pointer to a window. The function then creates a data structure (called a **control record**) of type `ControlRecord` from the information in the control resource, adds the control record to the control list for your window, and returns as its function result a handle to the control. (You use a control's handle when referring to the control in most other Control Manager routines; when you create scroll bars or pop-up menus for a window, you should store their handles in one of your application's own data structures for later reference.)

When you specify in the control resource that a control is initially visible and you use the `GetNewControl` function, the Control Manager uses the control's control definition function to draw the control inside its window. The Control Manager draws the control immediately, without using your window's standard updating mechanism. If you specify that a control is invisible, you can use the `ShowControl` procedure when you want to draw the control. Again, the Control Manager draws the control without using your window's standard updating mechanism. (Of course, even when the Control Manager draws the control, it might be completely or partially obscured from the user by overlapping windows or other objects.)

When your application receives an update event for a window that contains controls, you use the `UpdateControls` procedure in your application's standard window-updating code to redraw all the controls in the update region of the window.

Note
When you use the Dialog Manager to implement buttons, radio buttons, checkboxes, or pop-up menus in alert boxes and dialog boxes, Dialog Manager routines automatically use Control Manager routines to create and update these controls for you. If you implement any controls other than buttons, radio buttons, checkboxes, and pop-up menus in alert or dialog boxes—and whenever you implement *any* controls (scroll bars, for example) in your application's windows—you must explicitly use either the `GetNewControl` or the `NewControl` function to create the controls. You must always use the `UpdateControls` procedure to update controls you put in your own windows. ◆

When you use the Window Manager procedure `DisposeWindow` or `CloseWindow` to remove a window, either procedure automatically removes all controls associated with the window and releases the memory they occupy.

When you no longer need a control in a window that you want to keep, you can use the `DisposeControl` procedure, described on page 5-108, to remove it from the screen, delete it from its window's control list, and release the control record and all other associated data structures from memory. You can use the `KillControls` procedure, described on page 5-108, to dispose of all of a window's controls at once.

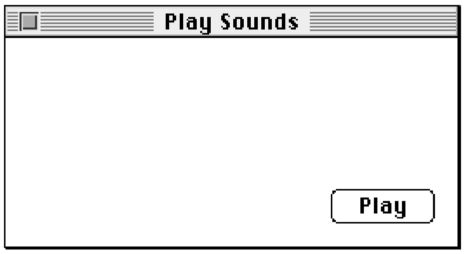
The next section, "Creating a Button, Checkbox, or Radio Button," provides a general discussion of the control resource as well as a more detailed description of the use of the control resource to specify buttons, checkboxes, and radio buttons in your application's windows. The two following sections, "Creating Scroll Bars" (beginning on page 5-21) and "Creating a Pop-Up Menu" (beginning on page 5-25), describe those elements of the control resource that differ from the control resources for buttons, checkboxes, and radio buttons. "Updating a Control" beginning on page 5-29 then offers an example of how you can use the `UpdateControls` procedure within your window-updating code.

Note
For the Control Manager to draw a control properly inside a window, the window must have its upper-left corner at local coordinates (0,0). If you use the QuickDraw procedure `SetOrigin` to change a window's local coordinate system, be sure to change it back—so that the upper-left corner is again at (0,0)—before drawing any of its controls. Because many Control Manager routines can (at least potentially) redraw a control, the safest policy after changing a window's local coordinate system is to change the coordinate system back before calling any Control Manager routine. ◆

Creating a Button, Checkbox, or Radio Button

Figure 5-10 shows a simple example of a button placed in a window of type `noGrowDocProc`—which you normally use to create a modeless dialog box. Although you usually use the Dialog Manager to create dialog boxes and their buttons, sometimes you might use the Window Manager and the Control Manager instead. The chapter "Dialog Manager" in this book explains why the use of the Window and Control Managers is sometimes preferable for this purpose.

Figure 5-10 A button in a simple window



Listing 5-1 shows an application-defined routine, `MyCreatePlaySoundsWindow`, that uses the `GetNewControl` function to create the button shown in Figure 5-10.

Listing 5-1 Creating a button for a window

```
FUNCTION MyCreatePlaySoundsWindow: OSErr;
VAR
   myWindow: WindowPtr;
BEGIN
   MyCreatePlaySoundsWindow := noErr;
   myWindow := GetNewWindow(rPlaySoundsModelessWindow, NIL, POINTER(-1));
   IF myWindow <> NIL THEN
   BEGIN
      {use the window's refCon to identify this window}
      SetWRefCon(myWindow, LongInt(kMyPlaySoundsWindow));
      SetPort(myWindow);
      gMyPlayButtonCtlHandle := GetNewControl(rPlayButton, myWindow);
      IF (gMyPlayButtonCtlHandle = NIL) THEN
         MyCreatePlaySoundsWindow := kControlErr;
   END
   ELSE
      MyCreatePlaySoundsWindow := kNoSoundWindow;
END;
```

The `MyCreatePlaySoundsWindow` routine begins by using the Window Manager function `GetNewWindow` to create a window; a pointer to that window is passed to `GetNewControl`. Note that, as explained in the chapter "Dialog Manager" in this book, you could create a modeless dialog box more easily by using the Dialog Manager function `GetNewDialog` and specifying its controls in an item list (`'DITL'`) resource.

For the resource ID of a control resource, the `MyCreatePlaySoundsWindow` routine defines an `rPlayButton` constant, which it passes to the `GetNewControl` function. Listing 5-2 shows how this control resource appears in Rez input format.

Listing 5-2 Rez input for a control resource

```
resource 'CNTL' (rPlayButton, preload, purgeable) {
   {87, 187, 107, 247},    /*rectangle*/
   0,                      /*initial setting*/
   visible,                /*make control visible*/
   1,                      /*maximum setting*/
   0,                      /*minimum setting*/
   pushButProc,            /*control definition ID*/
   0,                      /*reference value*/
   "Play"                  /*title*/
};
```

You supply the following information in the control resource for a button, checkbox, radio button, or scroll bar:

- a rectangle, specified by coordinates local to the window, that determines the control's size and location
- the initial setting for the control
- a constant (either `visible` or `invisible`) that specifies whether the control should be drawn on the screen immediately
- the maximum setting for the control
- the minimum setting for the control
- the control definition ID
- a reference value, which your application may use for any purpose
- the title of the control; or, for scroll bars, an empty string

As explained in "Creating a Pop-Up Menu" beginning on page 5-25, the values you supply in a control resource for a pop-up menu differ from those you specify for other buttons, checkboxes, radio buttons, and scroll bars.

Buttons are drawn to fit the rectangle exactly. To allow for the tallest characters in the system font, there should be at least a 20-point difference between the top and bottom coordinates of the rectangle. Listing 5-2 uses a rectangle with coordinates (87,187,107,247) to describe the size and location of the control within the window. Remember that the Control Manager will not draw controls properly unless the upper-left corner of the window coincides with the coordinates (0,0).

In Listing 5-2, the initial and minimum settings for the button are 0 and the maximum setting is 1. In control resources for buttons, checkboxes, and radio buttons, supply these values as the initial settings:

- For buttons, which don't retain a setting, specify a value of 0 for the initial and minimum settings and 1 for the maximum setting.
- For checkboxes and radio buttons, which retain an on-or-off setting, specify a value of 0 when you want to the control to be initially off. To turn a checkbox or radio button on, assign it an initial setting of 1. In response, the Control Manager places an X in a checkbox or a black dot in a radio button.

Because the `visible` identifier is specified in this example, the control is drawn immediately in its window. If you use the `invisible` identifier, your control is not drawn until your application uses the `ShowControl` procedure. When you want to make a visible control invisible, you can use the `HideControl` procedure.

In Listing 5-2, the maximum setting for the button is 1, which you, too, should specify in your control resources as the maximum setting for buttons, checkboxes, and radio buttons. In Listing 5-2, the minimum setting for the button is 0, which you, too, should specify in your control resources as the minimum setting for buttons, checkboxes, and radio buttons.

In Listing 5-2, the `pushButProc` constant is used to specify the control definition ID. Use the `checkBoxProc` constant to specify a checkbox and the `radioButProc` constant to specify a radio button.

Listing 5-2 specifies a reference value of 0. Your application can use this value for any purpose (except when you add the `popupUseAddResMenu` variation code to the `popupMenuProc` control definition function, as described in "Creating a Pop-Up Menu" beginning on page 5-25).

Finally, the string `"Play"` is specified as the title of the control. Buttons, checkboxes, and radio buttons require a title that communicates their purpose to the user. (The chapter "Dialog Manager" in this book offers extensive guidelines on appropriate titles for buttons.)

When specifying a title, make sure it fits in the control's rectangle; otherwise, the Control Manager truncates the title. For example, it truncates the titles of checkboxes and radio buttons on the right in Roman scripts, and it centers and truncates both ends of button titles.

If you localize your application for use with worldwide versions of system software, the titles may become longer or shorter. Translated text is often 50 percent longer than U.S. English text. You may need to resize your controls to accommodate the translated text.

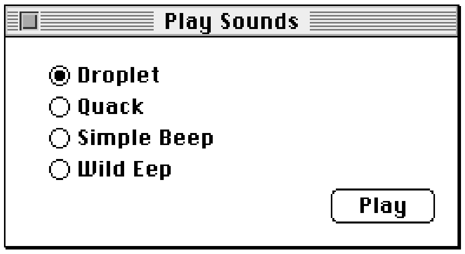
By default, the Control Manager displays control titles in the system font. To make it easier to localize your application for use with worldwide versions of system software, you should not change the font. Do not use a smaller font, such as 9-point Geneva; some script systems, such as KanjiTalk, require 12-point fonts. You can spare yourself future localization effort by leaving all control titles in the system font.

Follow book-title style when you capitalize control titles. In general, capitalize one-word titles and capitalize nouns, adjectives, verbs, and prepositions of four or more letters in multiple-word titles. You usually don't capitalize words such as *in, an,* or *and.* For capitalization rules, see the *Apple Publications Style Guide,* available from APDA.

The Control Manager allows button, checkbox, and radio button titles of multiple lines. When specifying a multiple-line title, end each line with the ASCII character code $0D (carriage return). If the control is a button, each line is horizontally centered, and the font leading is inserted between lines. (The height of each line is equal to the distance from the ascent line to the descent line plus the leading of the font used. Be sure to make the total height of the rectangle greater than the number of lines times this height.) If the control is a checkbox or a radio button, the text is justified as appropriate for the user's current script system, and the checkbox or button is vertically centered within its rectangle.

Figure 5-11 shows the Play Sounds window with four additional controls: radio buttons titled Droplet, Quack, Simple Beep, and Wild Eep.

Figure 5-11 Radio buttons in a simple window



Only one of these radio buttons can be on at a time. Listing 5-3 initially sets the Droplet radio button to 1, turning it on by default. This listing also shows the control resources for the other buttons, all initially set to 0 to turn them off.

For a checkbox or a radio button, always allow at least a 16-point difference between the top and bottom coordinates of its rectangle to accommodate the tallest characters in the system font.

Listing 5-3 Rez input for the control resources of radio buttons

```
resource 'CNTL' (cDroplet, preload, purgeable) {
   {13, 23, 31, 142},/*rectangle of control*/
   1,                /*initial setting*/
   visible,          /*make control visible*/
   1,                /*maximum setting*/
   0,                /*minimum setting*/
   radioButProc,     /*control definition ID*/
   0,                /*reference value*/
   "Droplet"         /*control title*/
};
resource 'CNTL' (cQuack, preload, purgeable) {
   {31, 23, 49, 142},/*rectangle of control*/
   0,                /*initial setting*/
   visible, 1, 0, radioButProc, 0, "Quack"};

resource 'CNTL' (cSimpleBeep, preload, purgeable) {
   {49, 23, 67, 142},/*rectangle of control*/
   0,                /*initial setting*/
   visible, 1, 0, radioButProc, 0, "Simple Beep"};

resource 'CNTL' (cWildEep, preload, purgeable) {
   {67, 23, 85, 142},/*rectangle of control*/
   0,                /*initial setting*/
   visible, 1, 0, radioButProc, 0, "Wild Eep"};
```

Creating Scroll Bars

When you define the control resource for a scroll bar, specify the `scrollBarProc` constant for the control definition ID. Typically, you make the scroll bar invisible and specify an initial value of 0, a minimum value of 0, and a maximum value of 0, and you supply an empty string for the title.

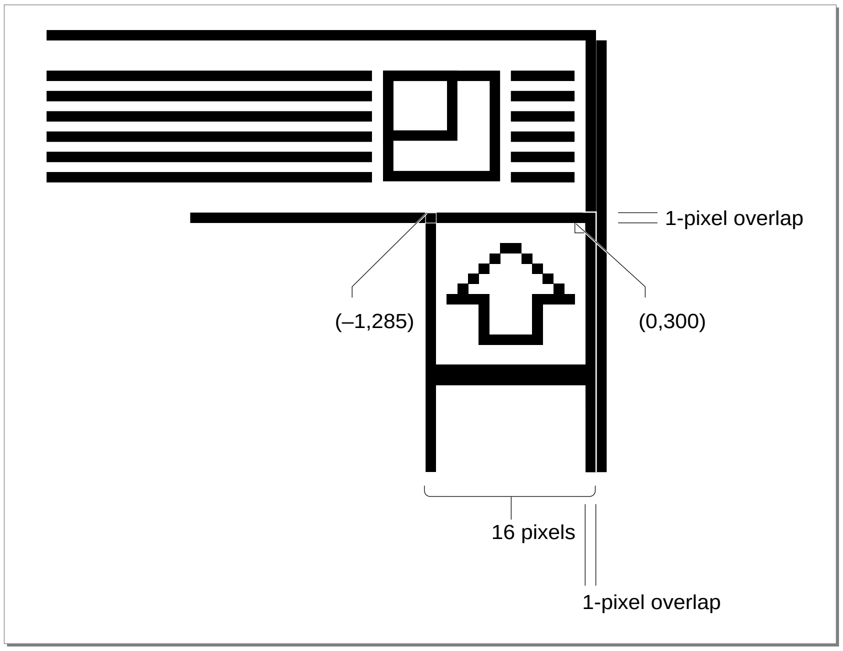
After you create a window, use the `GetNewControl` function to create the scroll bar you've defined in the control resource and to attach that scroll bar to the window. Use the `MoveControl`, `SizeControl`, `SetControlMaximum`, and `SetControlValue` procedures to adjust the location, size, and settings of the scroll bars, and then use the `ShowControl` procedure to display the scroll bars.

In your window-handling code, make the maximum setting the maximum area you want to allow the user to scroll. Most applications allow the user to drag the size box and click the zoom box to change the size of windows, and they allow the user to add information to and remove it from documents. To allow users to perform these actions, your application needs to calculate a changing maximum setting based upon the document's current size and its window's current size. For new documents that have no content to scroll to, assign an initial value of 0 as the maximum setting in the control resource; the control definition function automatically makes a scroll bar inactive when its minimum and maximum settings are identical. Thereafter, your window-handling routines should set and maintain the maximum setting, as described in "Determining and Changing Control Settings" beginning on page 5-37.

By convention, a scroll bar is 16 pixels wide, so there should be a 16-point difference between the left and right coordinates of a vertical scroll bar's rectangle and between the top and bottom coordinates of a horizontal scroll bar's rectangle. (If you don't provide a 16-pixel width, the Control Manager scales the scroll bar to fit the width you specify.) A standard scroll bar should be at least 48 pixels long, to allow room for the scroll arrows and scroll box.

The Control Manager draws lines that are 1 pixel wide for the rectangle enclosing the scroll bar. As shown in Figure 5-12, the outside lines of a scroll bar should overlap the lines that the Window Manager draws for the window frame.

Figure 5-12 How a scroll bar should overlap the window frame



To determine the rectangle for a *vertical* scroll bar, perform the following calculations and use their results in your control resource. (Do not include the area of the title bar in your calculations.)

- top coordinate = combined height of any items above the scroll bar – 1
- left coordinate = width of window – 15
- bottom coordinate = height of window – 14
- right coordinate = width of window + 1

To determine the rectangle for a *horizontal* scroll bar, perform the following calculations and use their results in your control resource.

- top coordinate = height of window – 15
- left coordinate = combined width of any items to the left of the scroll bar – 1
- bottom coordinate = height of window + 1
- right coordinate = width of window – 14

The top coordinate of a vertical scroll bar is –1, and the left coordinate of a horizontal scroll bar is –1, unless your application uses part of the window's typical scroll bar areas (in particular, those areas opposite the size box) for displaying information or specifying additional controls. For example, your application may choose to display the current page number of a document in the lower-left corner of the window—that is, in a small area to the left of its window's horizontal scroll bar. See *Macintosh Human Interface Guidelines* for a discussion of appropriate uses of a window's scroll bar areas for additional items and controls.

Just as the maximum settings of a window's scroll bars change when the user resizes the document's window, so too do the scroll bars' coordinate locations change when the user resizes the window. Although you must specify an initial maximum setting and location in the control resource for a scroll bar, your application must be able to change them dynamically—typically, by storing handles to each scroll bar in a document record when you create a window, and then by using Control Manager routines to change control settings (as described in "Determining and Changing Control Settings" beginning on page 5-37) and sizes and locations of controls (as described in "Moving and Resizing Scroll Bars" beginning on page 5-65).

Listing 5-4 shows a window resource (described in the chapter "Window Manager" in this book) for creating a window, and two control resources for creating the window's vertical and horizontal scroll bars. The rectangle for the initial size and shape of the window is specified in global coordinates, of course, and the rectangles for the two scroll bars are specified in coordinates local to the window.

Listing 5-4 Rez input for resources for a window and its scroll bars

```
                           /*initial window*/
resource 'WIND' (rDocWindow, preload, purgeable) {
   {64, 60, 314, 460},     /*initial rectangle for window*/
   zoomDocProc, invisible, goAway, 0x0, "untitled"
};
                           /*initial vertical scroll bar*/
resource 'CNTL' (rVScroll, preload, purgeable) {
   {-1, 385, 236, 401},    /*initial rectangle for control*/
      /*initial setting, visibility, max, min, ID, refcon, title*/
   0, invisible, 0, 0, scrollBarProc, 0, ""
};
```

```
                           /*initial horizontal scroll bar*/
resource 'CNTL' (rHScroll, preload, purgeable) {
   {235, -1, 251, 386},    /*initial rectangle for control*/
      /*initial setting, visibility, max, min, ID, refcon, title*/
   0, invisible, 0, 0, scrollBarProc, 0, ""
};
```

Listing 5-5 shows an application-defined procedure called `DoNew` that uses the `GetNewWindow` and `GetNewControl` functions to create a window and its scroll bars from the resources in Listing 5-4.

Listing 5-5 Creating a document window with scroll bars

```
PROCEDURE DoNew (newDocument: Boolean; VAR window: WindowPtr);
VAR
   good:                Boolean;
   windStorage:         Ptr;
   myData:              MyDocRecHnd;
BEGIN
   {use GetNewWindow or GetNewCWindow to create the window here}
   myData := MyDocRecHnd(NewHandle(SIZEOF(MyDocRec))); {create document rec}
   {test for errors along the way; if there are none, create the scroll }
   { bars and save their handles in myData}
   IF good THEN
   BEGIN    {create the vertical scroll bar and save its handle}
      myData^^.vScrollBar := GetNewControl(rVScroll, window);
      {create the horizontal scroll bar and save its handle}
      myData^^.hScrollBar := GetNewControl(rHScroll, window);
      good := (vScrollBar <> NIL) AND (hScrollBar <> NIL);
   END;
   IF good THEN
   BEGIN    {adjust size, location, settings, and visibility of scroll bars}
      MyAdjustScrollBars(window, FALSE);
      {perform other initialization here}
      IF NOT newDocument THEN
         ShowWindow(window);
   END;
   {clean up here}
END; {DoNew}
```

The `DoNew` routine uses Window Manager routines to create a window; its window resource specifies that the window is invisible. The window resource specifies an initial size and location for the window, but because the window is invisible, this window is not drawn.

Then `DoNew` creates a document record and stores a handle to it in the `myData` variable. The SurfWriter sample application uses this document record to store the data that the user creates in this window—as well as handles to the scroll bars that it creates. The SurfWriter sample application later uses these control handles to handle scrolling through the document and to move and resize the scroll bars when the user resizes the window. (See the chapter "Window Manager" in this book for more information about creating such a document record.)

To create scroll bars, `DoNew` uses `GetNewControl` twice—once for the vertical scroll bar and once for the horizontal scroll bar. The `GetNewControl` function returns a control handle; `DoNew` stores these handles in the `vScrollBar` and `hScrollBar` fields of its document record for later reference.

Because the window and the scroll bars are invisible, nothing is drawn onscreen yet for the user. Before drawing the window and its scroll bars, `DoNew` calls another application-defined procedure, `MyAdjustScrollBars`. In turn, `MyAdjustScrollBars` calls other application-defined routines that move and resize the scroll bars to fit the window and then calculate the maximum settings of these controls. (Listing 5-14 on page 5-39 shows the `MyAdjustScrollBars` procedure.)

After creating the window and its scroll bars, and then sizing and positioning them appropriately, `DoNew` uses the Window Manager procedure `ShowWindow` to display the window with its scroll bars.

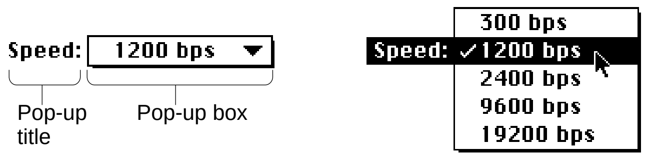
Creating a Pop-Up Menu

The values you specify in a control resource for a pop-up menu differ from those you specify for other controls. The control resource for a pop-up menu contains the following information:

- a rectangle, specified by coordinates local to the window, that determines the size and location of the pop-up title and pop-up box
- the alignment of the pop-up title with the pop-up box
- a constant (either `visible` or `invisible`) that specifies whether the control should be drawn on the screen immediately
- the width of the pop-up title
- the resource ID of the `'MENU'` resource describing the pop-up menu items
- the control definition ID
- a reference value, which your application may use for any purpose
- the title of the control

Figure 5-13 on the next page shows a pop-up menu; Listing 5-6 shows the control resource that creates this pop-up menu. (The chapter "Menu Manager" in this book recommends typical uses of pop-up menus and describes the relation between pop-up menus and menus you display in the menu bar.)

Figure 5-13 A pop-up menu

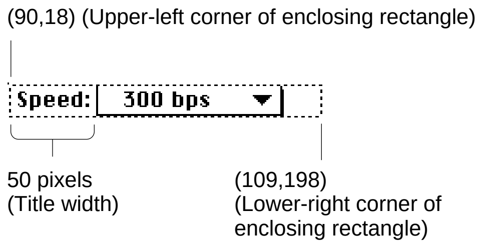


Listing 5-6 Rez input for the control resource of a pop-up menu

```
resource 'CNTL' (kPopUpCNTL, preload, purgeable) {
   {90, 18, 109, 198},  /*rectangle of control*/
   popupTitleLeftJust,  /*title position*/
   visible,             /*make control visible*/
   50,                  /*pixel width of title*/
   kPopUpMenu,          /*'MENU' resource ID*/
   popupMenuCDEFProc,   /*control definition ID*/
   0,                   /*reference value*/
   "Speed:"             /*control title*/
};
```

Listing 5-6 specifies a rectangle with the coordinates (90,18,109,198). Figure 5-14 illustrates the rectangle for this pop-up menu.

Figure 5-14 Dimensions of a sample pop-up menu


(90,18) (Upper-left corner of enclosing rectangle)
Speed:
300 bps
50 pixels (Title width)
(109,198) (Lower-right corner of enclosing rectangle)

Listing 5-6 uses the `popupTitleLeftJust` constant to specify the position of the control title. Specify any combination of the following constants (or their values) to inform the Control Manager where and how to draw the pop-up menu's title:

Setting	Constant	Description
$0000	`popupTitleLeftJust`	Place title left of the pop-up box
$0001	`popupTitleCenterJust`	Center title over the pop-up box
$00FF	`popupTitleRightJust`	Place title right of the pop-up box
$0100	`popupTitleBold`	Use boldface font style
$0200	`popupTitleItalic`	Use italic font style

Setting	Constant	Description
$0400	`popupTitleUnderline`	Use underline font style
$0800	`popupTitleOutline`	Use outline font style
$1000	`popupTitleShadow`	Use shadow font style
$2000	`popupTitleCondense`	Use condensed characters
$4000	`popupTitleExtend`	Use extended characters
$8000	`popupTitleNoStyle`	Use monostyle font

If `GetNewControl` completes successfully, it sets the value of the `contrlValue` field of the control record by assigning to that field the item number of the first menu item. When the user chooses a different menu item, the Control Manager changes the `contrlValue` field to that item number.

When you create pop-up menus, your application should store the handles for them; for example, in a record pointed to by the `refCon` field of a window record or a dialog record. (See the chapters "Window Manager" and "Dialog Manager" in this book for more information about the window record and the dialog record.) Storing these handles, as shown in the following code fragment, allows your application to respond later to users' choices in pop-up menus:

```
myData: MyDocRecHnd;
window: WindowPtr;

myData^^.popUpControlHandle := GetNewControl(kPopUpCNTL, window);
```

Listing 5-6 specifies 50 pixels (in place of a maximum setting) as the width of the control title. After it creates the control, the Control Manager sets the maximum value in the pop-up menu's control record to the number of items in the pop-up menu. Figure 5-14 illustrates this title width for the pop-up menu.

Listing 5-6 uses a `kPopUpMenu` constant to specify the resource ID of a `'MENU'` resource (in place of a minimum setting for the control). (See the chapter "Menu Manager" in this book for a description of the `'MENU'` resource type.) After it creates the control, the Control Manager assigns 1 as the minimum setting in the pop-up menu's control record.

IMPORTANT
When using the ResEdit application, version 2.1.1, you must use the same resource ID when specifying the menu resource and the control resource that together define a pop-up menu. ▲

You can also specify a different control definition ID by adding any or all of the following constants (or the variation codes they represent) to the `popupMenuProc` constant:

```
CONST popupFixedWidth      = $0001; {use fixed-width control}
      popupUseAddResMenu   = $0004; {use resource for menu items}
      popupUseWFont        = $0008; {use window font}
```

Constant	Description
`popUpFixedWidth`	Uses a constant control width. If your application specifies this value, the pop-up control definition function does not resize the control horizontally to fit long menu items. The width of the pop-up box is set to the width of the control, minus the width of the pop-up title your application specifies when it creates the control. If a menu item in a pop-up box does not fit in the space provided, the text is truncated to fit, and three ellipsis points (...) are appended at the end. If you do not specify this variation code, the pop-up control definition function may resize the control horizontally.
`popupUseAddResMenu`	Gets menu items from a resource other than the `'MENU'` resource. If your application specifies this value when creating a pop-up menu, the control definition function interprets the value in the `contrlRfCon` field of the control record as a value of type `ResType`. The control definition function uses the Menu Manager procedure `AppendResMenu` to add resources of that type to the menu.
`popupUseWFont`	Uses the font of the specified window. If your application specifies this value, the pop-up control definition function draws the pop-up menu title using the font and size of the window containing the control instead of using the system font.

The reference value that you specify in the control resource (and stored by the Control Manager in the `contrlRfCon` field of the control record) is available for your application's use. However, if you specify `popupUseAddResMenu` as a variation code, the Control Manager coerces the value in the `contrlRfCon` field of the control record to the type `ResType` and then uses `AppendResMenu` to add items of that type to the pop-up menu. For example, if you specify a reference value of `LongInt('FONT')` as the reference value, the control definition function appends a list of the fonts installed in the system to the menu associated with the pop-up menu. After the control has been created, your application can use the control record's `contrlRfCon` field for whatever use it requires. You can determine which menu item is currently chosen by calling `GetControlValue`.

Whenever the pop-up menu is redrawn, its control definition function calls the Menu Manager procedure `CalcMenuSize`. This procedure recalculates the size of the menu associated with the control (to allow for the addition or deletion of items in the menu). The pop-up control definition function may also update the width of the pop-up menu to the sum of the width of the pop-up title, the width of the longest item in the menu, the width of the downward-pointing arrow, and a small amount of white space. As previously described, your application can override this behavior by adding the variation code `popupFixedWidth` to the pop-up control definition ID.

You should not use the Menu Manager function `GetMenuHandle` to obtain a handle to a menu associated with a pop-up control. If necessary, you can obtain the menu handle (and the menu ID) of a pop-up menu by dereferencing the `contrlData` field of the pop-up menu's control record. The `contrlData` field of a control record is a handle to a block of private information. For pop-up menu controls, this field is a handle to a pop-up private data record, which is described on page 5-77.

Updating a Control

Your program should use the `UpdateControls` procedure upon receiving an update event for a window that contains controls such as scroll bars. (Window Manager routines such as `SelectWindow`, `ShowWindow`, and `BringToFront` do not automatically call `UpdateControls` to display the window's controls. Instead, they merely add the appropriate regions to the window's update region. This in turn generates an update event.)

Note
The Dialog Manager automatically updates the controls you use in alert boxes and dialog boxes. ◆

When your application receives an update event for a window that contains controls, use the `UpdateControls` procedure in your window-updating code to redraw all the controls in the update region of the window. Call `UpdateControls` after using the Window Manager procedure `BeginUpdate` and before using the Window Manager procedure `EndUpdate`.

When you call `UpdateControls`, you pass it parameters specifying the window to be updated and the window area that needs updating. Use the visible region of the window's graphics port, as referenced in the port's `visRgn` field, to specify the window's update region.

Listing 5-7 shows an application-defined routine, `DoUpdate`, that responds to an update event. The `DoUpdate` routine calls the Window Manager procedure `BeginUpdate`. To redraw this portion of the window, `DoUpdate` then calls another of its own procedures, `MyDrawWindow`.

Listing 5-7 Responding to an update event for a window

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:
      BEGIN
         BeginUpdate(window);
         MyDrawWindow(window);
         EndUpdate(window);
      END;  {of updating document windows}
   {handle other window types——modeless dialogs, etc.——here}
   END;  {of windowType CASE}
END;  {of DoUpdate}
```

Listing 5-8 illustrates how the SurfWriter sample application updates window controls and other window contents by using its own application-defined routine, `MyDrawWindow`. To draw only those controls in the window's update region, `MyDrawWindow` calls `UpdateControls`. To draw the size box in the lower-right corner of the window, `MyDrawWindow` calls the Window Manager procedure `DrawGrowIcon`. Finally, `MyDrawWindow` redraws the appropriate information contained in the user's document. Because the SurfWriter application uses TextEdit for all text editing in the window contents, Listing 5-8 calls the TextEdit procedure `TEUpdate`. (TextEdit is described in detail in *Inside Macintosh: Text*.)

Listing 5-8 Redrawing the controls in the update region

```
PROCEDURE MyDrawWindow (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN                {draw the contents of the window}
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
   BEGIN
      EraseRect(portRect);
      UpdateControls(window, visRgn);
      DrawGrowIcon(window);
      TEUpdate(portRect, myData^^.editRec);  {redraw text}
   END;
   HUnLock(Handle(myData));
END;  {MyDrawWindow}
```

For more information about updating window contents, see the chapter "Window Manager" in this book.

Responding to Mouse Events in a Control

The Control Manager provides several routines to help you detect and respond to mouse events involving controls. For mouse events in controls, you generally perform the following tasks:

1. In your event-handling code, use the Window Manager function `FindWindow` to determine the window in which the mouse-down event occurred.
2. If the mouse-down event occurred in the content region of your application's active window, use the `FindControl` function to determine whether the mouse-down event occurred in an active control and, if so, which control.
3. Call `TrackControl` to handle user interaction for the control for as long as the user holds the mouse button down. For scroll arrows and the gray areas of scroll bars, you

must define an action procedure for `TrackControl` to use. This action procedure should cause the document to scroll as long as the user holds down the mouse button. For pop-up menus, you pass `Pointer(-1)` in a parameter to `TrackControl` to use the action procedure defined in the pop-up control definition function. For the scroll box in scroll bars and for the other standard controls, you pass `NIL` in a parameter to `TrackControl` to get the Control Manager's standard response to mouse-down events.

4. When `TrackControl` reports that the user has released the mouse button with the cursor in a control, respond appropriately. This may require you to use other Control Manager routines, such as `GetControlValue` and `SetControlValue`, to determine and change control settings.

These and other routines for responding to events involving controls are described in the next several sections.

Note
The Dialog Manager procedure `ModalDialog` automatically calls `FindWindow`, `FindControl`, and `TrackControl` for mouse-down events in the controls of alert and modal dialog boxes. You can use the Dialog Manager function `DialogSelect`, which automatically calls `FindWindow`, `FindControl`, and `TrackControl`, to help you handle mouse events in your movable modal and modeless dialog boxes. ◆

Determining a Mouse-Down Event in a Control

When your application receives a mouse-down event, use the Window Manager function `FindWindow` to determine the window in which the event occurred. If the cursor was in the content region of your application's active window when the user pressed the mouse button, use the `FindControl` function to determine whether the mouse-down event occurred in an active control and, if so, which control.

When the mouse-down event occurs in a visible, active control, `FindControl` returns a handle to that control as well as a part code identifying the control's part. (Note that when the mouse-down event occurs in an invisible or inactive control, or when the cursor is not in a control, `FindControl` sets the control handle to `NIL` and returns 0 as its part code.)

A simple control such as a button or checkbox might have just one "part"; a more complex control can have as many parts as are needed to define how the control operates. A scroll bar has five parts: two scroll arrows, the scroll box, and the two gray areas on either side of the scroll box. Figure 5-4 on page 5-7 shows the five parts of a scroll bar.

A **part code** is an integer from 1 through 253 that identifies a part of a control. To allow different parts of a multipart control to respond to mouse events in different ways, many of the Control Manager routines accept a part code as a parameter or return one as a result. Part codes are assigned to a control by its control definition function. The standard control definition functions define the following part codes. Also listed are the constants you can use to represent them.

Constant	Part code	Control part
`inButton`	10	Button
`inCheckBox`	11	Entire checkbox or radio button
`inUpButton`	20	Up scroll arrow for a vertical scroll bar, left scroll arrow for a horizontal scroll bar
`inDownButton`	21	Down scroll arrow for a vertical scroll bar, right scroll arrow for a horizontal scroll bar
`inPageUp`	22	Gray area above scroll box for a vertical scroll bar, gray area to left of scroll box for a horizontal scroll bar
`inPageDown`	23	Gray area below scroll box for a vertical scroll bar, gray area to right of scroll box for a horizontal scroll bar
`inThumb`	129	Scroll box

The pop-up control definition function does not define part codes for pop-up menus. Instead (as explained in "Creating a Pop-Up Menu" beginning on page 5-25), your application should store the handles for your pop-up menus when you create them. Your application should then test the handles you store against the handles returned by `FindControl` before responding to users' choices in pop-up menus; this is described in more detail later in the next section.

Listing 5-9 illustrates an application-defined procedure, `DoMouseDown`, that an application might call in response to a mouse-down event. The `DoMouseDown` routine first calls the Window Manager function `FindWindow`, which returns two values: a pointer to the window in which the mouse-down event occurred and a constant that provides additional information about the location of that event. If `FindWindow` returns the `inContent` constant, then the mouse-down event occurred in the content area of one of the application's windows.

Listing 5-9 Detecting mouse-down events in a window

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:       Integer;
   thisWindow: WindowPtr;
BEGIN       {handle mouse-down event}
   part := FindWindow(event.where, thisWindow);
   CASE part OF
      inMenuBar:
         ;  {mouse-down in menu bar, respond appropriately here}
      inContent:
         IF thisWindow <> FrontWindow THEN
            {mouse-down in an inactive window; use SelectWindow }
            { to make it active here}
         ELSE        {mouse-down in the active window}
            DoContentClick(thisWindow, event);
      {handle other cases here}
   END; {of CASE statement}
END;  {DoMouseDown}
```

In Listing 5-9, when `FindWindow` reports a mouse-down event in the content region of a window containing controls, `DoMouseDown` calls another application-defined procedure, `DoContentClick`, and passes it the window pointer returned by the `FindWindow` function as well as the event record.

Listing 5-10 shows an application-defined procedure, `DoContentClick`, that uses this information to determine whether the mouse-down event occurred in a control.

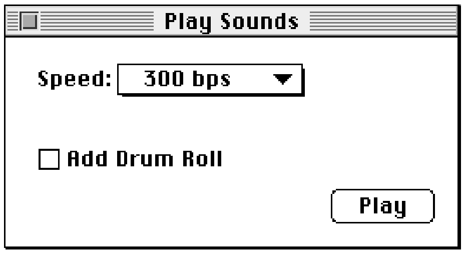
Listing 5-10 Detecting mouse-down events in a pop-up menu and a button

```
PROCEDURE DoContentClick (window: WindowPtr; event: EventRecord);
VAR
   mouse:      Point;
   control:    ControlHandle;
   part:       Integer;
   windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);    {get window type}

   CASE windowType OF

   kPlaySoundsModelessDialogBox:
      BEGIN
         SetPort(window);
         mouse := event.where;   {get the mouse location}
         GlobalToLocal(mouse);   {convert to local coordinates}
         part := FindControl(mouse, window, control);
         IF control = gSpeedPopUpControlHandle THEN
            {mouse-down in Modem Speed pop-up menu}
            DoPopUpMenu(mouse, control);
         CASE part OF
            inButton:   {mouse-down in Play button}
               DoPlayButton(mouse, control);
            inCheckBox: {mouse-down in checkbox}
               DoDrumRollCheckBox(mouse, control);
            OTHERWISE
            ;
         END;  {of CASE for control part codes}
      END;  {of kPlaySoundsModelessDialogBox case}
   {handle other window types, such as document windows, here}
   END; {of CASE for window types}
END; {of DoContentClick}
```

Figure 5-15 shows the Play Sounds window; `DoContentClick` uses the `FindControl` function to determine whether the mouse-down event occurred in the pop-up menu, the Play button, or the Add Drum Roll checkbox.

First, however, `DoContentClick` uses the event record to determine the cursor location, which is specified in global coordinates. Because the `FindControl` function expects the cursor location in coordinates local to the window, `DoContentClick` uses the QuickDraw procedure `GlobalToLocal` to convert the point stored in the `where` field of the event record to coordinates local to the current window. The `GlobalToLocal` procedure takes one parameter, a point in global coordinates—where the upper-left corner of the entire bit image is coordinate (0,0). See *Inside Macintosh: Imaging* for more information about the `GlobalToLocal` procedure.

Figure 5-15 Three controls in a window



When it calls `FindControl`, `DoContentClick` passes the cursor location in the window's local coordinates as well as the pointer returned earlier by the `FindWindow` function (shown in Listing 5-9 on page 5-32).

If the cursor is in a control, `FindControl` returns a handle to the control and a part code indicating the control part. Because the pop-up control definition function does not define control parts, `DoContentClick` tests the control handle returned by `FindControl` against a pop-up menu's control handle that the application stores in its own global variable. If these are handles to the same control, `DoContentClick` calls another application-defined routine, `DoPopUpMenu`.

After checking whether `FindControl` returns a control handle to a pop-up menu, `DoContentClick` uses the part code that `FindControl` returns to determine whether the cursor is in one of the other two controls. If `FindControl` returns the `inButton` constant, `DoContentClick` calls another application-defined routine, `DoPlayButton`. If `FindControl` returns the `inCheckBox` constant, `DoContentClick` calls another application-defined routine, `DoDrumRollCheckBox`.

As described in the next section, all three of these application-defined routines—`DoPopUpMenu`, `DoPlayButton`, and `DoDrumRollCheckBox`—in turn use the `TrackControl` function to follow and respond to the user's mouse movements in the control reported by `FindControl`.

Tracking the Cursor in a Control

After using the `FindControl` function to determine that the user pressed the mouse button when the cursor was in a control, use the `TrackControl` function first to follow and respond to the user's mouse movements, and then to determine which control part contains the cursor when the user releases the mouse button.

Generally, you use `TrackControl` after using the `FindControl` function to determine that the mouse-down event occurred in a control. You pass to `TrackControl` the control handle returned by the `FindControl` function, and you also pass to `TrackControl` the same point you passed to `FindControl` (that is, a point in coordinates local to the window).

The `TrackControl` function follows the movements of the cursor in a control and provides visual feedback until the user releases the mouse button. The visual feedback given by `TrackControl` depends on the control part in which the mouse-down event occurred. When highlighting the control is appropriate—in a button, for example—`TrackControl` highlights the control part (and removes the highlighting when the user releases the mouse button). When the user presses the mouse button while the cursor is in an indicator (such as the scroll box of a scroll bar) and then moves the mouse, `TrackControl` responds by dragging a dotted outline of the indicator. Figure 5-8 on page 5-12 illustrates how `TrackControl` provides visual feedback.

You can also use an action procedure to undertake additional actions as long as the user holds down the mouse button. For example, if the user is working in a text document and holds down the mouse button while the cursor is in a scroll arrow, your action procedure should continuously scroll through the document one line (or some equivalent measure) at a time until the user releases the button or reaches the end of the document. You pass a pointer to this procedure to `TrackControl`. ("Scrolling in Response to Events in Scroll Arrows and Gray Areas" beginning on page 5-57 describes how to do this.)

The `TrackControl` function returns the control's part code if the user releases the mouse button while the cursor is inside the control part, or 0 if the user releases the mouse button while the cursor is outside the control part. Unless `TrackControl` returns 0 as its function result, your application should then respond as appropriate to a mouse-up event in that control part. When `TrackControl` returns 0 as its function result, your application should do nothing.

Listing 5-11 on the next page shows an application-defined procedure, `DoPlayButton`, that uses `TrackControl` to track mouse-down events in the Play button shown in Figure 5-15. The `DoPlayButton` routine passes, to `TrackControl`, the control handle returned by `FindControl`. The `DoPlayButton` routine also passes to `TrackControl` the same cursor location it passed to `FindControl` (that is, a point in local coordinates). Because buttons don't need an action procedure, `NIL` is passed as the final parameter to `TrackControl`.

Listing 5-11 Using the `TrackControl` function with a button

```
PROCEDURE DoPlayButton (mouse: Point; control: ControlHandle);
BEGIN
   IF TrackControl(control, mouse, NIL) <> 0 THEN  {user clicks Play}
   BEGIN
      IF gPlayDrumRoll = TRUE THEN  {user clicked Play Drum Roll checkbox }
         DoPlayDrumRoll;            { so play a drum roll first}
      SysBeep(30);   {always play system alert sound when user clicks Play}
   END;
END;
```

When the user presses the mouse button when the cursor is in the Play button, `TrackControl` inverts the Play button. If the user releases the mouse button after moving the cursor outside the control part, `TrackControl` stops inverting the button and returns the value 0, in which case `DoPlayButton` does nothing.

If, however, the user releases the mouse button with the cursor in the Play button, `TrackControl` stops inverting the Play button and returns the value for the `inButton` constant. Then `DoPlayButton` calls the Sound Manager procedure `SysBeep` to play the system alert sound (which is described in the chapter "Dialog Manager" in this book). Before releasing the mouse button, the user can move the cursor away from the control part and then return to it, and `TrackControl` will still return the part code when the user releases the mouse button.

For buttons, checkboxes, radio buttons, and the scroll box in a scroll bar, your application typically passes `NIL` to `TrackControl` to use no action procedure. However, `TrackControl` still responds visually to mouse events in active controls. That is, when the user presses the mouse button with the cursor over a control whose action procedure is set to `NIL`, `TrackControl` changes the control's display appropriately until the user releases the mouse button.

For scroll arrows and for the gray areas of a scroll box, you need to define your own action procedures. You pass a pointer to the action procedure as one of the parameters to `TrackControl`, as described in "Scrolling in Response to Events in Scroll Arrows and Gray Areas" beginning on page 5-57.

For a pop-up menu, you must pass `Pointer(-1)` to `TrackControl` for its action procedure; this causes `TrackControl` to use the action procedure defined in the pop-up control definition function.

Listing 5-10 on page 5-33 calls an application-defined routine, `DoPopUpMenu`, when `FindControl` reports a mouse-down event in a pop-up menu. Listing 5-12 shows how `DoPopUpMenu` uses `TrackControl` to handle user interaction in the pop-up menu. By passing `Pointer(-1)` to `TrackControl`, `DoPopUpMenu` uses the action procedure defined in the pop-up control definition function.

Listing 5-12 Using `TrackControl` with a pop-up menu

```
PROCEDURE DoPopUpMenu (mouse: Point; control: ControlHandle);
VAR
   menuItem:   Integer;
   part:       Integer;
BEGIN
   part := TrackControl(control, mouse, Pointer(-1));
   menuItem := GetControlValue(control);
   IF menuItem <> gCurrentItem THEN
   BEGIN
      gCurrentItem := menuItem;
      SetMyCommunicationSpeed; {use speed stored in gCurrentItem}
   END;
END; {of DoPopUpMenu}
```

The action procedure for pop-up menus highlights the pop-up menu title, displays the pop-up menu, and handles all user interaction while the user drags up and down the menu. When the user releases the mouse button, the action procedure closes the pop-up box, draws the user's choice in the pop-up box (or restores the previous item if the user doesn't make a new choice), and removes the highlighting of the pop-up title. The pop-up control definition function then changes the value of the `contrlValue` field of the control record to the number of the menu item chosen by the user.

Because buttons do not retain settings, responding to them is very straightforward: when the user clicks a button, your application should immediately undertake the action described by the button's title. For pop-up menus and other types of controls, you must determine their current settings before responding to the user's action. For example, before responding, you need to know which item the user has chosen in a pop-up menu, whether a checkbox is checked, or how far the user has moved the scroll box. The action you take may, in turn, involve changing other control settings. Determining and changing control settings are described in the next section.

After learning how to determine and change control settings, see "Scrolling Through a Document" beginning on page 5-43 for a detailed discussion of how to respond to mouse events in scroll bars.

Determining and Changing Control Settings

Using either the control resource or the parameters to the `NewControl` function, your application specifies a control's various default values—such as its current setting and minimum and maximum settings—when it creates the control.

When the user clicks a control, however, your application often needs to determine the current setting and other possible values of that control. When the user clicks a checkbox, for example, your application must determine whether the box is checked before your application can decide whether to clear or draw a checkmark inside the checkbox. When the user moves the scroll box, your application needs to determine what part of the document to display.

Applications must adjust some controls in response to events other than mouse events in the controls themselves. For example, when the user resizes a window, your application must use the Control Manager procedures `MoveControl` and `SizeControl` to move and resize the scroll bars appropriately.

Your application can use the `GetControlValue` function to determine the current setting of a control, and it can use the `GetControlMaximum` function to determine a control's maximum setting.

You can use the `SetControlValue` procedure to change the control's setting and redraw the control accordingly. You can use the `SetControlMaximum` procedure to change a control's maximum setting and to redraw the indicator or scroll box to reflect the new setting.

In response to user action involving a control, your application often needs to change the setting and possibly redraw the control. When the user clicks a checkbox, for example, your application must determine whether the checkbox is currently selected or not, and then switch its setting. When you use `SetControlValue` to switch a checkbox setting, the Control Manager either draws or removes the X inside the checkbox, as appropriate. When the user clicks a radio button, your application must determine whether the radio button is already on and, if not, turn the previously selected radio button off and turn the newly selected radio button on.

Figure 5-15 on page 5-34 shows a checkbox in the Play Sounds window. When the user clicks the checkbox to turn it on, the application adds a drum roll to the sound it plays whenever the user clicks the Play button.

Listing 5-13 shows the application-defined routine `DoDrumRollCheckBox`, which responds to a click in a checkbox. This routine uses the `GetControlValue` function to determine the last value of the checkbox and then uses the `SetControlValue` procedure to change it. The `GetControlValue` function returns a control's current setting, which is stored in the `contrlValue` field of the control record. The `SetControlValue` procedure sets the `contrlValue` field to the specified value and redraws the control to reflect the new setting. (For checkboxes and radio buttons, the value 1 fills the control with the appropriate mark, and the value 0 removes the mark. For scroll bars, `SetControlValue` redraws the scroll box at the appropriate position along the scroll bar. For a pop-up menu, `SetControlValue` displays in its pop-up box the name of the menu item corresponding to the specified value.)

Listing 5-13 Responding to a click in a checkbox

```
PROCEDURE DoDrumRollCheckBox (mouse: Point; control: ControlHandle);
VAR
   checkbox:Integer;
BEGIN
   IF TrackControl(control, mouse, NIL) <> 0 THEN  {user clicks checkbox}
   BEGIN
      checkbox := GetControlValue(control);  {get last value of checkbox}
      checkbox := 1 - checkbox;              {toggle value of checkbox}
      SetControlValue(control, checkbox);    {set checkbox to new value}
      IF checkbox = 1 THEN                   {the checkbox is checked}
         gPlayDrumRoll := TRUE {play a drum roll next time user clicks Play}
      ELSE
         gPlayDrumRoll := FALSE;
   END;
END;
```

The `DoDrumRollCheckBox` routine uses `TrackControl` to determine which control the user selects. When `TrackControl` reports that the user clicks the checkbox, `DoDrumRollCheckBox` uses `GetControlValue` to determine whether the user last selected the checkbox (that is, whether the control has a current setting of 1) or deselected it (in which case, the control has a current setting of 0). By subtracting the control's current setting from 1, `DoDrumRollCheckBox` toggles to a new setting and then uses `SetControlValue` to assign this new setting to the checkbox. The `SetControlValue` procedure changes the current setting of the checkbox and redraws it appropriately, by either drawing an X in the box if the new setting of the control is 1 or removing the X if the new setting of the control is 0.

Listing 5-4 on page 5-23 shows the control resources that specify a window's scroll bars, and Listing 5-5 on page 5-24 shows an application's `DoNew` routine for creating a document window with these scroll bars. This routine uses the `GetNewControl` function to create the scroll bars and then calls an application-defined routine, `MyAdjustScrollBars`. Listing 5-14 shows `MyAdjustScrollBars`, which in turn calls other application-defined routines that determine the proper sizes, locations, and maximum settings of the scroll bars.

Listing 5-14 Adjusting scroll bar settings and locations

```
PROCEDURE MyAdjustScrollBars (window: WindowPtr;
                              resizeScrollBars: Boolean);
VAR
   myData: MyDocRecHnd;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH myData^^ DO
   BEGIN
      HideControl(vScrollBar);   {hide the vertical scroll bar}
      HideControl(hScrollBar);   {hide the horizontal scroll bar}
      IF resizeScrollBars THEN   {move and size if needed}
         MyAdjustScrollSizes(window);
      MyAdjustScrollValues(window, NOT resizeScrollBars);
      ShowControl(vScrollBar);   {show the vertical scroll bar}
      ShowControl(hScrollBar);   {show the horizontal scroll bar}
   END;
   HUnLock(Handle(myData));
END; {of MyAdjustScrollbars}
```

When calling the `DoOpen` routine to open an existing document in a window, SurfWriter also uses this `MyAdjustScrollBars` procedure to size and adjust the scroll bars. When the user changes the window's size, the SurfWriter application uses `MyAdjustScrollBars` again.

The `MyAdjustScrollBars` routine begins by getting a handle to the window's document record, which stores handles to the scroll bars as well as other relevant data about the document. (See the chapter "Window Manager" in this book for information about creating your application's own document record for a window.)

Before making any adjustments to the scroll bars, `MyAdjustScrollBars` passes the handles to these controls to the Control Manager procedure `HideControl`, which makes the controls invisible. The `MyAdjustScrollBars` routine then calls another application-defined procedure, `MyAdjustScrollSizes` (shown in Listing 5-24 on page 5-67), to move and resize the scroll bars appropriately. After calling yet another application-defined procedure, `MyAdjustScrollValues`, to set appropriate current and maximum settings for the scroll bars, `MyAdjustScrollBars` uses the Control Manager procedure `ShowControl` to display the scroll bars in their new locations.

Listing 5-15 shows how the `MyAdjustScrollValues` procedure calls another application-defined routine, `MyAdjustHV`, which uses Control Manager routines to assign appropriate settings to the scroll bars.

Listing 5-15 Assigning settings to scroll bars

```
PROCEDURE MyAdjustScrollValues (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH myData^^ DO
   BEGIN
      MyAdjustHV(TRUE, vScrollBar, editRec);
      MyAdjustHV(FALSE, hScrollBar, editRec);
   END;
   HUnLock(Handle(myData));
END; {of MyAdjustScrollValues}
```

To prevent the user from scrolling past the edge of the document and seeing a blank window, you should limit the scroll bars' maximum settings, as illustrated in Figure 5-6 on page 5-10. If the window is larger than the document (which can easily happen with small documents on large monitors), your application should make the maximum scroll bar settings identical to their minimum settings. In this case, the Control Manager then makes the scroll bars inactive, which is appropriate when all the information fits in the window.

Listing 5-16 shows the application-defined `MyAdjustHV` procedure, used for adjusting the current and maximum settings for a scroll bar. When passed `TRUE` in the `isVert` parameter, `MyAdjustHV` calculates and adjusts the maximum and current settings for the vertical scroll bar; when passed `FALSE`, it calculates and adjusts those settings for the horizontal scroll bar.

In this example, the document consists of monostyled text stored in a TextEdit edit record. The `viewRect` field of a TextEdit edit record specifies the rectangle where the text is visible; because `viewRect` already excludes the scroll bar regions, `MyAdjustHV` does not need to subtract the scroll bar regions from the window height or width when calculating the maximum settings for these scroll bars. (For more information about TextEdit in general and the edit record in particular, see *Inside Macintosh: Text.*)

Listing 5-16 Adjusting the maximum and current settings for a scroll bar

```
PROCEDURE MyAdjustHV (isVert: Boolean; control: ControlHandle;
                      editRec: TEHandle);
VAR
   oldValue, oldMax, width:   Integer;
   max, lines, value:         Integer;
BEGIN
   {calculate new maximum and current settings for the vertical or }
   { horizontal scroll bar}
   oldMax := GetControlMaximum(control);
   oldValue := GetControlValue(control);
   MyGetDocWidth(width);
   IF isVert THEN    {adjust max setting for the vertical scroll bar}
   BEGIN
      lines := editRec^^.nLines;
      {since nLines isn't right if the last character is a carriage }
      { return, check for that case}
      IF Ptr(ORD(editRec^^.hText^) + editRec^^.teLength - 1)^ = kCRChar THEN
         lines := lines + 1;
      max := lines - ((editRec^^.viewRect.bottom - editRec^^.viewRect.top)
                      DIV editRec^^.lineHeight);
   END
   ELSE           {adjust max setting for the horizontal scroll bar}
      max := width - (editRec^^.viewRect.right - editRec^^.viewRect.left);
   IF max < 0 THEN
      max := 0;   {check for negative settings}
   SetControlMaximum(control, max); {set the max value of the control}
   IF isVert THEN {adjust current setting for vertical scroll bar}
      value := (editRec^^.viewRect.top - editRec^^.destRect.top)
                DIV editRec^^.lineHeight
   ELSE           {adjust current setting for the horizontal scroll bar}
      value := editRec^^.viewRect.left - editRec^^.destRect.left;
   IF value < 0 THEN
      value := 0
   ELSE IF value > max THEN
      value := max;  {don't allow current setting to be greater than the }
                     { maximum setting}
   SetControlValue(control, value);
END; {of MyAdjustHV}
```

The `MyAdjustHV` routine first uses the `GetControlMaximum` and `GetControlValue` functions to determine the maximum and current settings for the scroll bar being adjusted.

Then `MyAdjustHV` calculates a new maximum setting for the case of a vertical scroll bar. Because the window displays a text-only document, `MyAdjustHV` uses the `nLines` field of the edit record to determine the total number of lines in—and hence, the length of—the document. Then `MyAdjustHV` subtracts the calculated height of the window from the length of the document, and makes this value the maximum setting for the vertical scroll bar.

To calculate the total height in pixels of the window, `MyAdjustHV` begins by subtracting the top coordinate of the view rectangle from its bottom coordinate. (The upper-left corner of a window is normally at point [0,0]; therefore the vertical coordinate of a point at the bottom of a rectangle has a larger value than a point at the top of the rectangle.) Then `MyAdjustHV` divides the pixel height of the window by the value of the edit record’s `lineHeight` field, which for monostyled text specifies the document’s line height in pixels. By dividing the window height by the line height of the text, `MyAdjustHV` determines the window’s height in terms of lines of text.

The `MyAdjustHV` routine uses another application-defined routine, `MyGetDocWidth`, to determine the width of the document. To calculate the width of the window, `MyAdjustHV` subtracts the left coordinate of the view rectangle from its right coordinate. By subtracting the window width from the document width, `MyAdjustHV` derives the maximum setting for the horizontal scroll bar.

For both vertical and horizontal scroll bars, `MyAdjustHV` assigns a maximum setting of 0 whenever the window is larger than the document—for instance, when a window is created for a new document that contains no data yet. In this case, `MyAdjustHV` assigns the same value, 0, to both the maximum and current settings for the scroll bar. The standard control definition function for scroll bars automatically makes a scroll bar inactive when its minimum and maximum settings are identical. This is entirely appropriate, because whenever the user has nowhere to scroll, the scroll bar should be inactive. When you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.

The `MyAdjustHV` routine then uses the Control Manager procedure `SetControlMaximum` to assign the newly calculated maximum settings to either scroll bar. The `SetControlMaximum` procedure revises the control to reflect the new maximum setting; for example, if the user deletes a large portion of the document, thereby reducing the maximum setting, `SetControlMaximum` moves the scroll box to indicate the new position relative to the smaller document.

When the user adds information to or removes information from a document or adjusts its window size, your application may need to adjust the current setting of the scroll bar as well. The `MyAdjustHV` routine calculates a new current setting for the control and then uses `SetControlValue` to assign that setting to the control as well as to reposition the scroll box accordingly.

The destination rectangle, specified in the `destRect` field of the edit record, is the rectangle in which the text is drawn, whereas the view rectangle is the rectangle in which the text is actually visible. By subtracting the top coordinate of the destination rectangle from the top coordinate of the view rectangle, and dividing the result by the line height, `MyAdjustHV` derives the number of the line currently displayed at the top of the window. This is the line number `MyAdjustHV` uses for the current setting of the vertical scroll bar.

To derive the current setting of the horizontal scroll bar in terms of pixels, `MyAdjustHV` subtracts the left coordinate of the destination rectangle from the left coordinate of the view rectangle.

Scrolling Through a Document

Earlier sections of this chapter explain how to create scroll bars, determine when a mouse-down event occurs in a scroll bar, track user actions in a scroll bar, and determine and change scroll bar settings. This section discusses how your application actually scrolls through documents in response to users' mouse activity in the scroll bars. For example, your application scrolls toward the bottom of the document under the following conditions:

- When the user drags the scroll box to the bottom of the vertical scroll bar, your application should display the end of the user's document.
- When the user clicks the gray area below the scroll box, your application should move the document up to display the next window of information toward the bottom of the document, and it should use `SetControlValue` to move the scroll box.
- When the user clicks the down scroll arrow, your application should move the document up by one line (or by some similar measure) and bring more of the bottom of the document into view, and it should use `SetControlValue` to move the scroll box.

As a first step, your application must determine the distance by which to scroll. When the user drags a scroll box to a new location on the scroll bar, you scroll a corresponding distance to a new location in the document.

When the user clicks a scroll arrow, your application determines an appropriate amount to scroll. In general, a word processor scrolls vertically by one line of text and horizontally by the average character width, and a database or spreadsheet scrolls by one field. Graphics applications should scroll to display an entire object when possible. (Typically, applications convert these distances to pixels when using Control Manager, QuickDraw, and TextEdit routines.)

When the user clicks a gray area of a scroll bar, your application should scroll by a distance of just less than the height or width of the window. To determine this height and width, you can use the `contrlOwner` field of the scroll bar's control record. This field contains a pointer to the window record. When you scroll by a distance of one window, it is best to retain part of the previous window. This retained portion helps the user place the material in context. For example, if the user scrolls down by a distance of one window in a text document, the line at the top of the window should be the one that previously appeared at the bottom of the window.

The scrolling direction is determined by whether the scrolling distance is expressed as a positive or negative number. When the user scrolls down or to the right, the scrolling distance is a negative number; when the user scrolls up or to the left, the scrolling distance is a positive number. For example, when the user scrolls from the beginning of a document to a line located 200 pixels down, the scrolling distance is –200 pixels on the vertical scroll bar. When the user scrolls from there back to the start of the document, the scrolling distance is 200 pixels.

Determining the scrolling distance is only the first step. In brief, your application should take the following steps to scroll through a document in response to the user's manipulation of a scroll bar.

1. Use the `FindControl`, `GetControlValue`, and `TrackControl` functions to help calculate the scrolling distance.
2. If you are scrolling for any reason other than the user dragging the scroll box, use the `SetControlValue` procedure to move the scroll box a corresponding amount.
3. Use a routine—such as the QuickDraw procedure `ScrollRect` or the TextEdit procedure `TEPinScroll`—to move the bits displayed in the window by the calculated scrolling distance. Then either use a call that generates an update event or else directly call your application's `DoUpdate` routine, which should perform the rest of these steps.
4. Use the `UpdateControls` procedure to update the scroll bars and then call the Window Manager procedure `DrawGrowIcon` to redraw the size box.
5. Use the QuickDraw procedure `SetOrigin` to change the window origin by an amount equal to the scroll bar settings so that the upper-left corner of the document lies at (0,0) in the window's local coordinate system. (You perform this step so that your application's document-drawing routines can draw in the correct area of the window.)
6. Call your application's routines for redrawing the document inside the window.
7. Use the `SetOrigin` procedure to reset the window origin to (0,0) so that future Window Manager and Control Manager routines draw in the correct area of the window.
8. Return to your event loop.

These steps are explained in greater detail in the rest of this section.

Note

It is not necessary to use `SetOrigin` as described in the rest of this chapter. This procedure merely helps you to offset the window origin by the scroll bars' current settings when you update the window, so that you can locate objects in a document using a coordinate system where the upper-left corner of the document is always at (0,0). As an alternative to this approach, your application can leave the upper-left corner of the window (called the **window origin**) located at (0,0) and instead offset the items in your document by an amount equal to the scroll bars' settings. The QuickDraw procedures `OffsetRect`, `OffsetRgn`, `SubPt`, and `AddPt`, which are described in *Inside Macintosh: Imaging,* are useful if you pursue this alternate approach. ◆

When the user saves a document, your application should store the data in your own application-defined data structures. (For example, the sample code in this chapter stores a handle to a TextEdit edit record in a *document record.* The edit record contains information about the text, such as it length and its own local coordinate system, and a handle to the text itself.) You typically store information about the objects your application displays onscreen by using coordinates local to the document, where the upper-left corner of the document is located at (0,0).

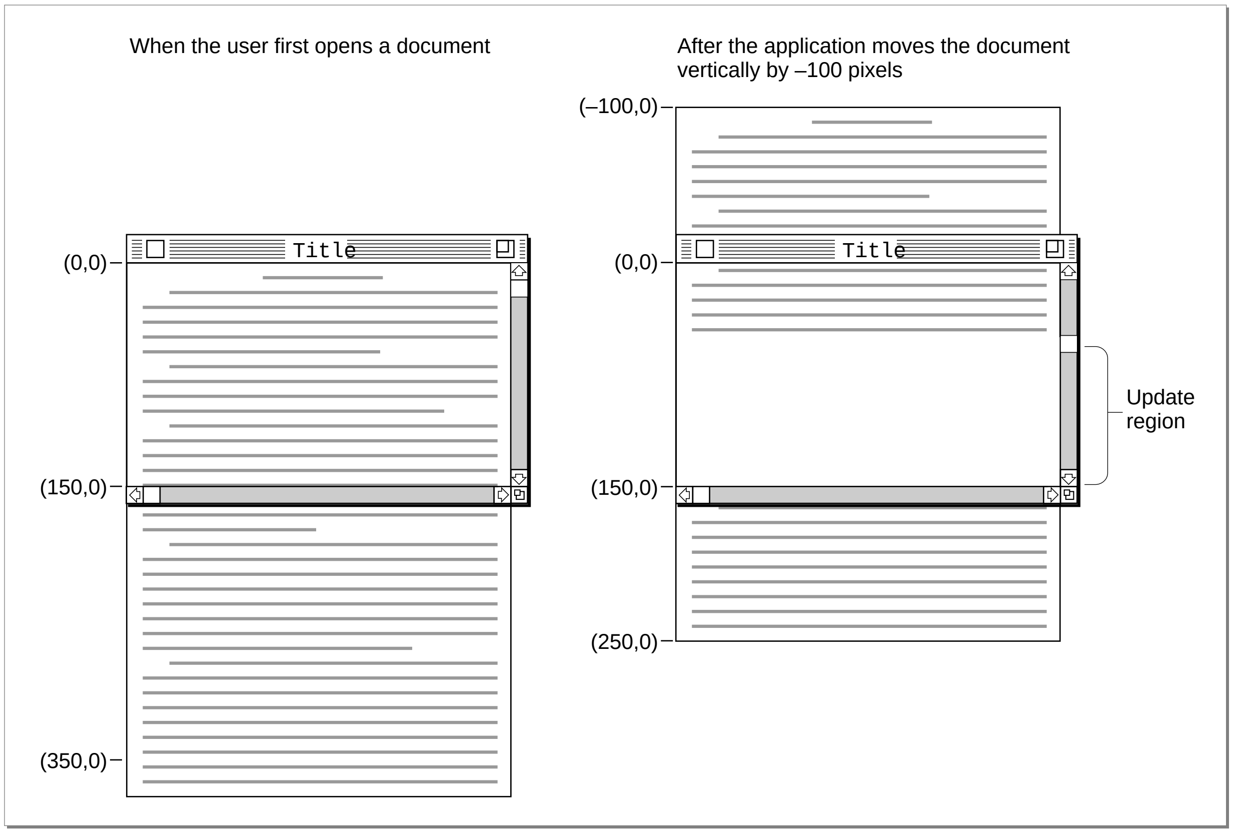
The left side of Figure 5-16 on the next page illustrates a case in which the user has just opened an existing document, and the SurfWriter sample application displays the top of the document. In this example, the document consists of 35 lines of monostyled text, and the line height throughout is 10 pixels. Therefore, the document is 350 pixels long. When the user first opens the document, the window origin is identical to the upper-left point of the document's space: both are at (0,0).

In this example, the window displays 15 lines of text, which amount to 150 pixels. Hence, the maximum setting for the scroll bar is 200 because the vertical scroll bar's maximum setting is the length of the document minus the height of its window.

Imagine that the user drags the scroll box halfway down the vertical scroll bar. Because the user wishes to scroll down, the SurfWriter application must move the text of the document up so that more of the bottom of the document shows. Moving a document *up* in response to a user request to scroll *down* requires a scrolling distance with a *negative* value. (Likewise, moving a document *down* in response to a user request to scroll *up* requires a scrolling distance with a *positive* value.)

Using `FindControl`, `TrackControl`, and `GetControlValue`, the SurfWriter application determines that it must move the document up by 100 pixels—that is, by a scrolling distance of –100 pixels. (Using `FindControl`, `TrackControl`, and `GetControlValue` to determine the scrolling distance is explained in detail in "Scrolling in Response to Events in the Scroll Box" beginning on page 5-53.)

Figure 5-16 Moving a document relative to its window


When the user first opens a document
After the application moves the document vertically by –100 pixels
(–100,0)
(0,0)
Title
(0,0)
Title
Update region
(150,0)
(150,0)
(250,0)
(350,0)

The SurfWriter application then uses the QuickDraw procedure `ScrollRect` to shift the bits displayed in the window by a distance of –100 pixels. The `ScrollRect` procedure moves the document upward by 100 pixels (that is, by 10 lines); 5 lines from the bottom of the previous window display now appear at the top of the window, and the SurfWriter application adds the rest of the window to an update region for later updating.

The `ScrollRect` procedure doesn't change the coordinate system of the window; instead it moves the bits in the window to new coordinates that are still in the window's local coordinate system. For purposes of updating the window, you can think of this as changing the coordinates of the entire document, as illustrated in the right side of Figure 5-16.

The `ScrollRect` procedure takes four parameters: a rectangle to scroll, a horizontal distance to scroll, a vertical distance to scroll, and a region handle. Typically, when specifying the rectangle to scroll, your application passes a value representing the content region minus the scroll bar regions, as shown in Listing 5-17.

Listing 5-17 Using `ScrollRect` to scroll the bits displayed in the window

```
PROCEDURE DoGraphicsScroll (window: WindowPtr;
                            hDistance, vDistance: Integer);
VAR
   myScrollRect: Rect;
   updateRegion: RgnHandle;
BEGIN
   {initially, use the window's portRect as the rectangle to scroll}
   myScrollRect := window^.portRect;
   {subtract vertical and horizontal scroll bars from rectangle}
   myScrollRect.right := myScrollRect.right - 15;
   myScrollRect.bottom := myScrollRect.bottom - 15;
   updateRegion := NewRgn;    {always initialize the update region}
   ScrollRect(myScrollRect, hDistance, vDistance, updateRegion);
   InvalRgn(updateRegion);
   DisposeRgn(updateRegion);
END; {of DoGraphicsScroll}
```

IMPORTANT
You must first pass a horizontal distance as a parameter to `ScrollRect` and then pass a vertical distance. Notice that when you specify a point in the QuickDraw coordinate system, the opposite is true: you name the vertical coordinate first and the horizontal coordinate second. ▲

Although each scroll bar is 16 pixels along its shorter dimension, the `DoGraphicsScroll` procedure shown in Listing 5-17 subtracts only 15 pixels because the edge of the scroll bar overlaps the edge of the window frame, leaving only 15 pixels of the scroll bar in the content region of the window.

The bits that `ScrollRect` shifts outside of the rectangle specified by `myScrollRect` are not drawn on the screen, and they are not saved—it is your application's responsibility to keep track of this data.

The `ScrollRect` procedure shifts the bits a distance of `hDistance` pixels horizontally and `vDistance` pixels vertically; when `DoGraphicsScroll` passes positive values in these parameters, `ScrollRect` shifts the bits in the `myScrollRect` parameter to the right and down, respectively. This is appropriate when the user intends to scroll left or up, because when the SurfWriter application finishes updating the window, the user sees more of the left and top of the document, respectively. (Remember: to scroll up or left, move the document down or right, both of which are in the positive direction.)

When `DoGraphicsScroll` passes negative values in these parameters, `ScrollRect` shifts the bits in the `myScrollRect` parameter to the left or up. This is appropriate when the user intends to scroll right or down, because when the SurfWriter application finishes updating the window, the user sees more of the right and the bottom of the document. (Remember: to scroll down or right, move the document up or left, both of which are in the negative direction.)

In Figure 5-16, the SurfWriter application determines a vertical scrolling distance of –100, which it passes in the `vDistance` parameter as shown here:

```
ScrollRect(myScrollRect, 0, –100, updateRegion);
```

If, however, the user were to move the scroll box back to the beginning of the document at this point, the SurfWriter application would determine that it has a distance of 100 pixels to scroll up, and it would therefore pass a positive value of 100 in the `vDistance` parameter.

After using `ScrollRect` to move the bits that already exist in the window, the SurfWriter application should draw the bits in the update region of the window by using its standard window-updating code.

As previously explained, `ScrollRect` in effect changes the coordinates of the document relative to the local coordinates of the window. In terms of the window's local coordinate system, the upper-left corner of the document is now at (–100, 0), as shown on the right side of Figure 5-16. To facilitate updating the window, the SurfWriter application uses the QuickDraw procedure `SetOrigin` to change the local coordinate system of the window so that the SurfWriter application can treat the upper-left corner of the document as again lying at (0,0).

The `SetOrigin` procedure takes two parameters: the first is a new horizontal coordinate for the window origin, and the second is a new vertical coordinate for the window origin.

IMPORTANT
Like `ScrollRect`, `SetOrigin` requires you to pass a horizontal coordinate and then a vertical coordinate. Notice that when you specify a point in the QuickDraw coordinate system, the opposite is true: you name the vertical coordinate first and the horizontal coordinate second. ▲

Any time you are ready to update a window (such as after scrolling it), you can use `GetControlValue` to determine the current setting of the horizontal scroll bar and pass this value as the new horizontal coordinate for the window origin. Then use `GetControlValue` to determine the current setting of the vertical scroll bar and pass this value as the new vertical coordinate for the window origin. Using `SetOrigin` in this fashion shifts the window's local coordinate system so that the upper-left corner of the document is always at (0,0) when you redraw the document within its window.

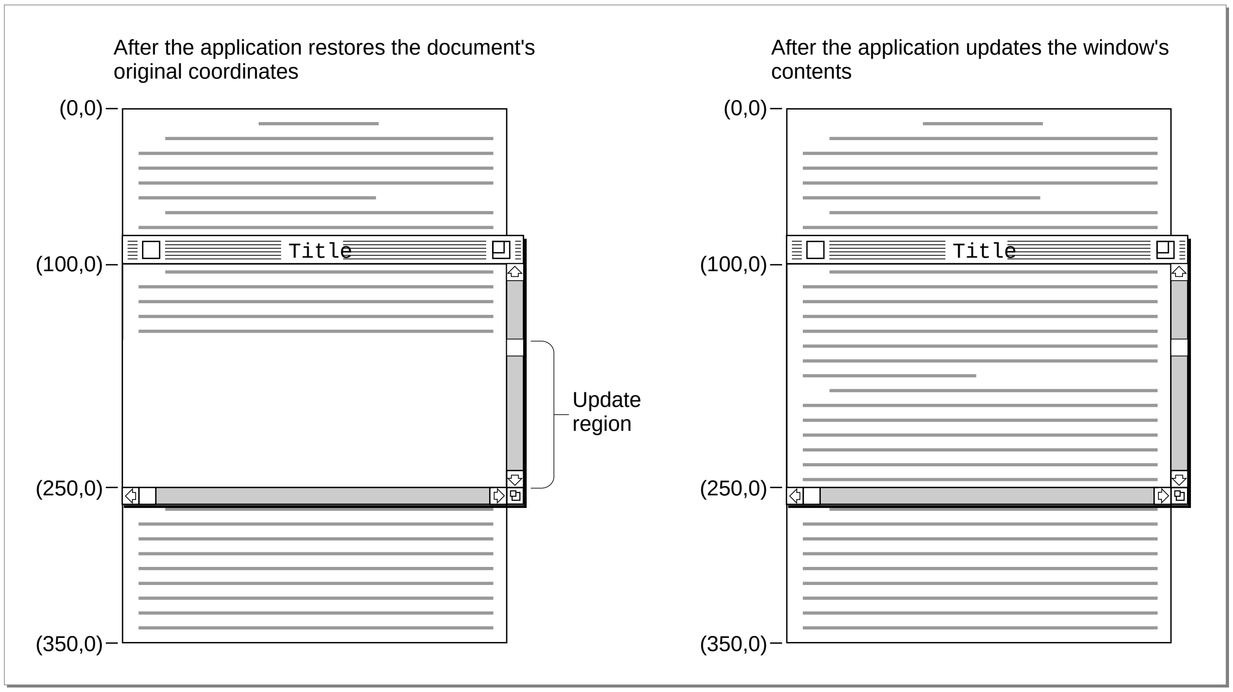
For example, after the user manipulates the vertical scroll bar to move (either up or down) to a location 100 pixels from the top of the document, the SurfWriter application makes the following call:

```
SetOrigin(0, 100);
```

Although the scrolling distance was –100, which is relative, the current setting for the scroll bar is now at 100. (Because you specify a point in the QuickDraw coordinate system by its vertical coordinate first and then its horizontal coordinate, the order of parameters to `SetOrigin` may be initially confusing.)

The left side of Figure 5-17 shows how the SurfWriter application uses the `SetOrigin` procedure to move the window origin to the point (100,0) so that the upper-left corner of the document is now at (0,0) in the window's local coordinate system. This restores the document's original coordinate space and makes it easier for the application to draw in the update region of the window.

Figure 5-17 Updating the contents of a scrolled window


After the application restores the document's original coordinates
(0,0)
(100,0)
(250,0)
(350,0)
Title
Update region
After the application updates the window's contents
(0,0)
(100,0)
(250,0)
(350,0)
Title

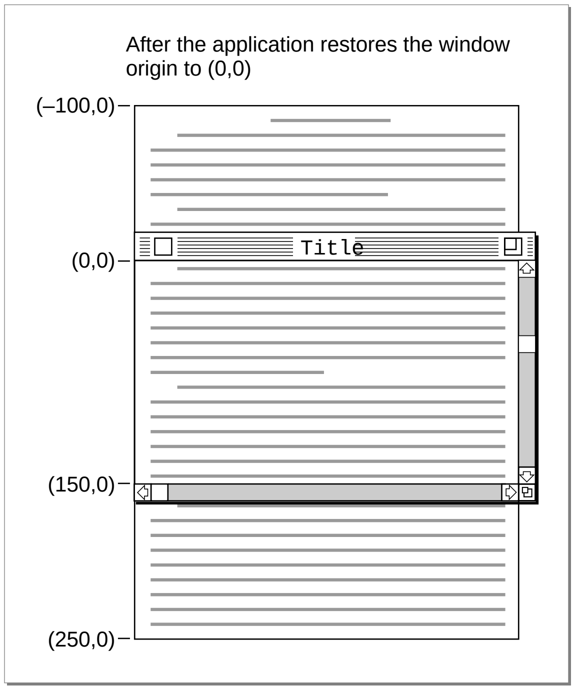
After restoring the document's original coordinates, the SurfWriter application updates the window, as shown on right side of Figure 5-17. The application draws lines 16 through 24, which it stores in its document record as beginning at (160,0) and ending at (250,0).

To review what has happened up to this point: the user has dragged the scroll box one-half of the distance down the vertical scroll bar; the SurfWriter application determines that this distance amounts to a scroll distance of –100 pixels; the SurfWriter application passes this distance to `ScrollRect`, which shifts the bits in the window 100 pixels upward and creates an update region for the rest of the window; the SurfWriter application passes the vertical scroll bar's current setting (100 pixels) in a parameter to `SetOrigin` so that the document's local coordinates are used when the update region of the window is redrawn; and, finally, the SurfWriter application draws the text in the update region of the window.

However, the window origin cannot be left at (100,0); instead, the SurfWriter application must use `SetOrigin` to reset it to (0,0) after performing its own drawing, because the

Window and Control Managers always assume the window's upper-left point is at (0,0) when they draw in a window. Figure 5-18 shows how the application uses `SetOrigin` to set the window origin back to (0,0) at the conclusion of its window-updating routine. After the update, the application begins processing events in its event loop again.

Figure 5-18 Restoring the window origin to (0,0)



The left side of Figure 5-19 illustrates what happens when the user scrolls all the way to the end of the document—a distance of another 10 lines, or 100 pixels. After the SurfWriter application calls `ScrollRect`, the bottom 5 lines from the previous window display appear at the top of the new window and the bottom of the window becomes a new update region. Because the user has scrolled a total distance of 200 pixels, the application uses `SetOrigin` to change the window origin to (200,0), as shown on the right side of Figure 5-19.

The left side of Figure 5-20 shows the SurfWriter application drawing in the update region of the window; the right side of the figure shows the SurfWriter application restoring the window origin to (0,0).

Figure 5-19 Scrolling to the end of a document

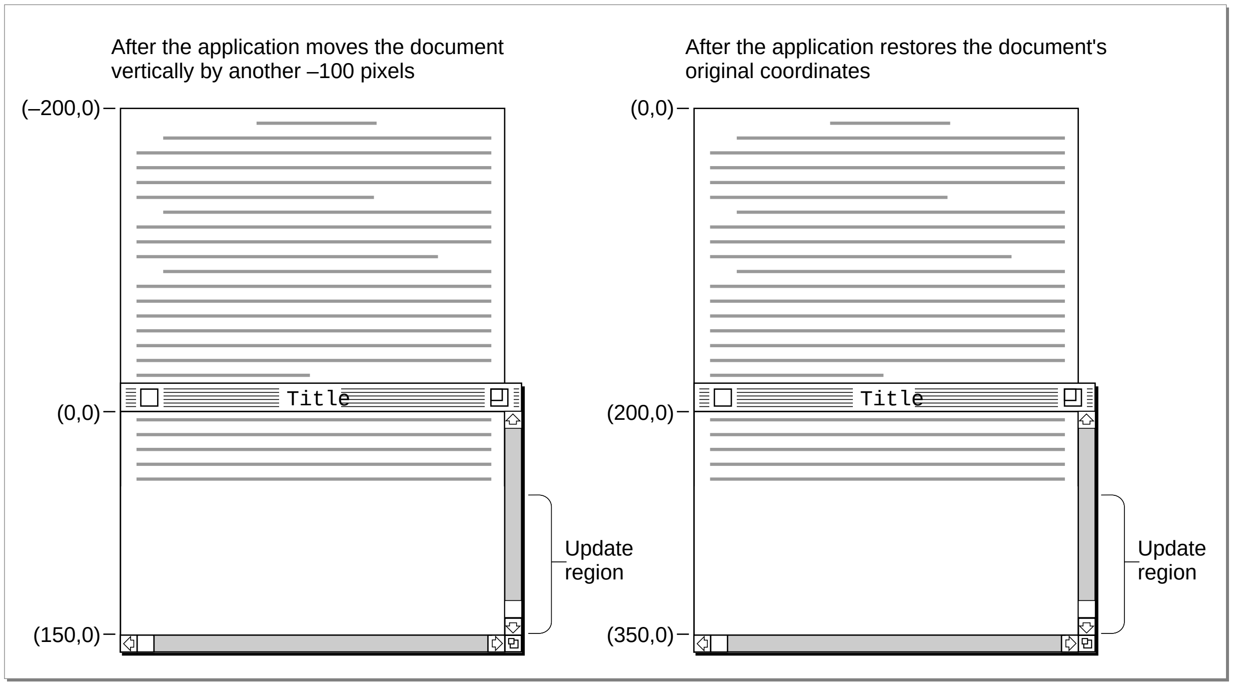
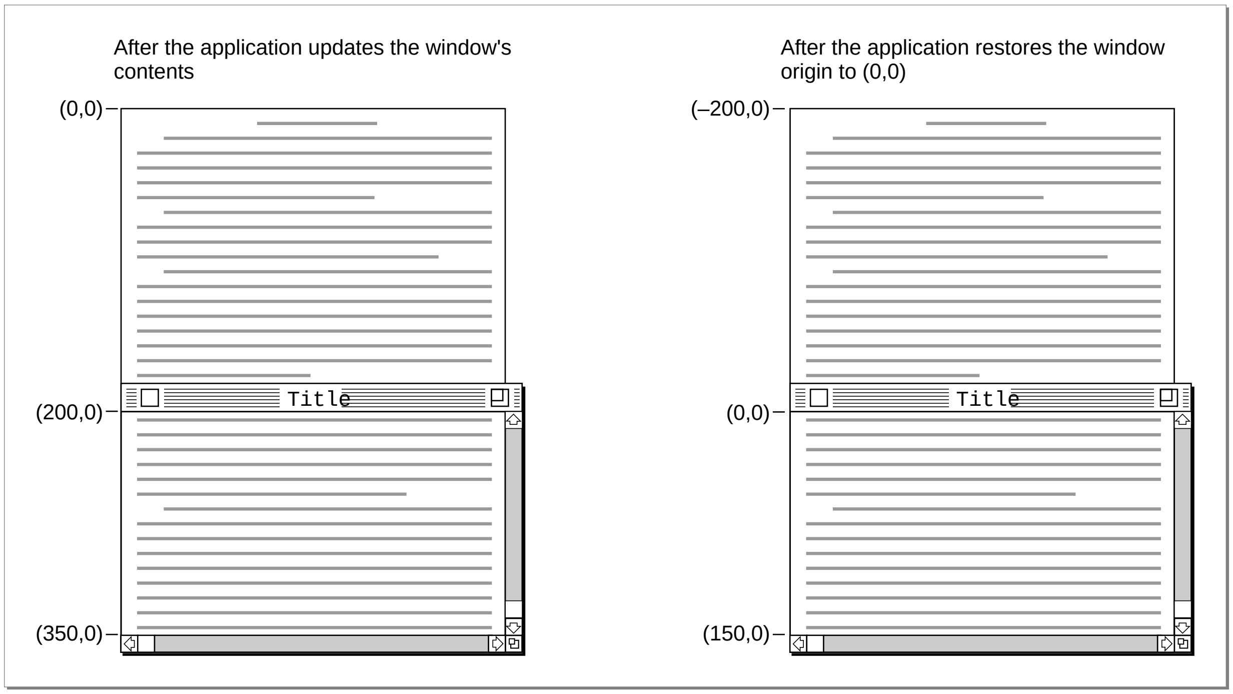

After the application moves the document vertically by another –100 pixels
(–200,0)
(0,0)
(150,0)
Title
Update region
After the application restores the document's original coordinates
(0,0)
(200,0)
(350,0)
Title
Update region

Figure 5-20 Updating a window’s contents and returning the window origin to (0,0)


After the application updates the window's contents
(0,0)
(200,0)
(350,0)
Title
After the application restores the window origin to (0,0)
(–200,0)
(0,0)
(150,0)
Title

How your application determines a scrolling distance and how it then moves the bits in the window by this distance are explained in greater detail in the next two sections, “Scrolling in Response to Events in the Scroll Box” and “Scrolling in Response to Events in Scroll Arrows and Gray Areas.” “Drawing a Scrolled Document Inside a Window,” which follows these two sections, describes what your application should do in its window-updating code to draw in a window that has been scrolled. You can find more detailed information about the `SetOrigin` and `ScrollRect` procedures in *Inside Macintosh: Imaging.*

So far, this discussion has assumed that you are scrolling in response to the user’s manipulation of a scroll bar. Most of the time, the user decides when and where to scroll. However, in addition to user manipulation of scroll bars, there are four cases in which your application must scroll through the document. Your application design must take these cases into account.

- When your application performs an operation whose side effect is to make a new selection or move the insertion point, you should scroll to show the new selection. For example, when the user invokes a search operation, your application locates the desired text. If this text appears in a part of the document that isn’t currently visible, you should scroll to show the selection. Such scrolling might also be necessary after the user invokes a paste operation. If the insertion point appears after the end of whatever was pasted, scroll until the selection and the new insertion point are visible.
- When the user enters information from the keyboard at the edge of a window, you should scroll to incorporate and display the new information. The user’s focus will be on the new information, so it doesn’t make sense to maintain the document’s position and record the new information out of the user’s view. In general, a word processor scrolls one line of text, and a database or spreadsheet scrolls one field. Graphics applications should scroll to display an entire object when possible. Otherwise, determine how quickly your application can redraw the window contents during scrolling and adjust the scrolling to minimize blinking and redrawing. Try to ensure that the scrolling is sufficiently fast so as not to annoy users but not so fast as to confuse them.
- When the user moves the cursor past the edge of the window while holding down the mouse button to make an extended selection, you should scroll the window in the direction of cursor movement. The rate of scrolling can be the same as if the user were holding down the mouse button on the corresponding scroll arrow. In some cases it makes sense to vary the scrolling speed so that it is faster as the user moves the cursor farther away from the edge of the window.
- Sometimes the user selects something, scrolls to a new location, and then tries to perform an operation on the selection. In this case, you should scroll so that the selection is showing before your application performs the operation. Showing the selection makes it clear to the user what is being changed.

When designing the document-scrolling routines for your application, also try to keep the following user interface guidelines in mind:

- Whenever your application scrolls automatically, avoid unnecessary scrolling. Users want to control the position of documents, so your application should move a document only as much as necessary. Thus, if part of a selection is already showing in a window, don’t scroll at all. One exception to this rule is when the hidden part of the selection is more important than the visible part; then scroll to show the important part. For example, suppose a user selects a large block of text and only the bottom is currently visible. If the user then types a character, your application must scroll to the location of the newly typed characters so that they are visible.

- If your application can scroll in one orientation to reveal the selection, don't scroll in both orientations. That is, if you can scroll vertically to show the selection, don't also scroll horizontally.
- When you can show context on either side of a selection, it's useful to do so. It's also better to position a selection somewhere near the middle of a window than against a corner. When the selection is too large to fit in the window, it's helpful to display unselected information at either the beginning or the end of the selection to provide context.

Scrolling in Response to Events in the Scroll Box

"Responding to Mouse Events in a Control" beginning on page 5-30 describes in general how to use `FindControl` and `TrackControl` in your event-handling code. Listing 5-18 shows how to use these routines to respond in particular to mouse events in a scroll bar.

Listing 5-18 Responding to mouse events in a scroll bar

```
PROCEDURE DoContentClick (window: WindowPtr; event: EventRecord);
VAR
   mouse:               Point;
   control:             ControlHandle;
   part:                Integer;
   myData:              MyDocRecHnd;
   oldSetting:          Integer;
   scrollDistance:      Integer;
   windowType:          Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(window));
            HLock(Handle(myData));
            mouse := event.where;
            GlobalToLocal(mouse);   {convert to local coordinates}
            part := FindControl(mouse, window, control);
            CASE part OF
               {handle all other parts first; handle scroll bar parts last}
            inThumb:    {mouse-down in scroll box}
               BEGIN    {get scroll bar setting}
                  oldSetting := GetControlValue(control);
                  {let user drag scroll box around}
                  part := TrackControl(control, mouse, NIL);
                  {until user releases mouse button}
                  IF part = inThumb THEN
                  BEGIN                {get new distance to scroll}
                     scrollDistance := oldSetting - GetControlValue(control);
                     IF scrollDistance <> 0 THEN
                        IF control = myData^^.vScrollBar THEN
                           TEPinScroll(0, scrollDistance *
                                       myData^^.editRec^^.lineHeight,
                                       myData^^.editRec);
                        ELSE
                           TEPinScroll(scrollDistance, 0, myData^^.editRec);
                  END; {of handling mouse-up in scroll box}
               END; {of handling mouse-down in scroll box}
            inUpButton, inDownButton, inPageUp, in PageDown:
            {mouse-down in scroll arrows or gray areas}
               IF control = myData^^.vScrollBar THEN
                        {handle vertical scroll}
                  part := TrackControl(control, mouse, @MyVerticalActionProc)
               ELSE     {handle horizontal scroll}
                  part := TrackControl(control, mouse, @MyHorzntlActionProc);
            OTHERWISE   ;
            END; {of CASE part}
            HUnLock(Handle(myData));
         END; {of kMyDocWindowType}
         {handle other window types here}
   END; {of CASE windowType}
END;
```

When the user presses the mouse button while the cursor is in a visible, active scroll box, `FindControl` returns as its result the part code for a scroll box. That part code and the constant you can use to represent it are listed here:

Constant	Part code	Control part
`inThumb`	129	Scroll box

As shown in Listing 5-18, when `FindControl` returns the value for `inThumb`, your application should immediately call `GetControlValue` to determine the current setting of the scroll bar. If the user drags the scroll box, you subtract from this setting the new current setting that becomes available when the user releases the mouse button, and you use this result for your scrolling distance.

After using `GetControlValue` to determine the current setting of the scroll bar, use `TrackControl` to follow the movements of the cursor inside the scroll box and to drag a dotted outline of the scroll box in response to the user's movements.

When the user releases the mouse button, `TrackControl` returns `inThumb` if the cursor is still in the scroll box or 0 if the cursor is outside the scroll box. When `TrackControl` returns 0, your application does nothing. Otherwise, your application again uses `GetControlValue` to calculate the distance to scroll.

Calculate the distance to scroll by calling `GetControlValue` and subtracting the new current setting of the scroll bar from its previous setting, which you determine by calling `GetControlValue` before the user releases the mouse button. If this distance is not 0, you should move the bits in the window by this distance and update the contents of the rest of the window.

Before scrolling, you must determine if the scroll bar is a vertical scroll bar or a horizontal scroll bar. As previously explained in this chapter, you should store handles to your scroll bars in a document record, one of which you create for every document. By comparing the field containing the vertical scroll bar handle, you can determine whether the control handle returned by `FindControl` is the handle to the vertical scroll bar. If so, the user has moved the scroll box of the vertical scroll box. If not, the user has moved the scroll box of the horizontal scroll bar.

After determining which scroll bar contains the scroll box that the user has dragged, you move the document contents of the window by the appropriate scrolling distance. That is, for a positive scrolling distance in the vertical scroll bar, move the bits in the window down by that distance. When you update the window, this displays more lines from the top of the document—which is appropriate when the user moves the scroll box *up.* For a positive scrolling distance in the horizontal scroll bar, move the bits in the window to the right by that distance. When you update the window, this displays more lines from the left side of the document—which is appropriate when the user moves the scroll box *to the left.* (Remember: to scroll up or left, move the document down or right, both of which are in the positive direction.)

For a negative scrolling distance in the vertical scroll bar (such as that shown in Figure 5-16 on page 5-46), move the bits in the window up by that distance. When you update the window, this displays more lines from the bottom of the document—which is appropriate when the user moves the scroll box *down.* For a negative scrolling distance in the horizontal scroll bar, move the bits in the window to the left by that distance. When you update the window, this displays more lines from the right side of the document—which is appropriate when the user moves the scroll box *to the right.* (Remember: to scroll down or right, move the document up or left, both of which are in the negative direction.)

The previous examples in this chapter have shown an application that uses a TextEdit edit record to store monostyled text created by the user. For simple text-handling needs, TextEdit provides many routines that simplify your work; for example, the `TEPinScroll` procedure scrolls through the text in the view rectangle of an edit record by the number of pixels specified by your application; `TEPinScroll` stops scrolling when the last line scrolls into the view rectangle.

The TEPinScroll procedure takes three parameters: the number of pixels to move the text horizontally, the number of pixels to move the text vertically, and a handle to an edit record. Positive values in the first two parameters move the text right and down, respectively, and negative values move the text left and up.

The DoContentClick procedure, illustrated in Listing 5-18 on page 5-53, passes the scrolling distance in the second parameter of TEPinScroll for a vertical scroll bar, and it passes the scrolling distance in the first parameter for a horizontal scroll bar.

Listing 5-16 on page 5-41 shows an application-defined routine, MyAdjustHV, called by the SurfWriter sample application whenever it creates, opens, or resizes a window. This routine defines the current and maximum settings for a vertical scroll bar in terms of lines of text.

The DoContentClick procedure on page 5-53 uses GetControlValue to determine the control's current setting—which for the vertical scroll bar DoContentClick calculates as some number of lines. When determining the vertical scroll bar's scrolling distance, DoContentClick again calculates a value representing some number of lines.

However, TEPinScroll expects pixels, not lines, to be passed in its parameters. Therefore, DoContentClick multiplies the scrolling distance (which it calculates as some number of lines of text) by the line height (which is maintained in the edit record for monostyled text as some number of pixels). In this way, DoContentClick passes a scrolling distance—in terms of pixels—to TEPinScroll, as shown in this code fragment.

```
IF control = myData^^.vScrollBar THEN
   TEPinScroll(0, scrollDistance * myData^^.editRec^^.lineHeight,
               myData^^.editRec);
```

Figure 5-16 on page 5-46 illustrates a scrolling distance of –10 lines. If the line height is 10 pixels, the SurfWriter application passes –100 as the second parameter to TEPinScroll.

The TEPinScroll procedure adds the scrolled-away area to the update region and generates an update event so that the text in the edit record's view rectangle can be updated. In its code that handles update events for windows, the SurfWriter sample application then uses the TEUpdate procedure—as described in "Drawing a Scrolled Document Inside a Window" beginning on page 5-62— for its windows that include TextEdit edit records.

To learn more about TEPinScroll, the TextEdit edit record, and other facilities offered by TextEdit, see *Inside Macintosh: Text.*

The QuickDraw procedure ScrollRect is a more general-purpose routine for moving bits in a window when scrolling. If you use ScrollRect to scroll the bits displayed in the window, you should define a routine like DoGraphicsScroll, shown in Listing 5-17 on page 5-47, and use it instead of TEPinScroll, which is used in Listing 5-18 on page 5-53.

The ScrollRect procedure returns in the updateRegion parameter the area that needs to be updated. The DoGraphicsScroll procedure shown in Listing 5-17 on page 5-47 then uses the QuickDraw procedure InvalRgn to add this area to the update region, forcing an update event. In your code for handling update events, you draw in the area of the window from which `ScrollRect` has moved the bits, as described in "Drawing a Scrolled Document Inside a Window" beginning on page 5-62.

When a mouse-down event occurs in the scroll arrows or gray areas of the vertical scroll bar, the `DoContentClick` routine in Listing 5-18 on page 5-53 calls `TrackControl` and passes it a pointer to an application-defined action procedure called `MyVerticalActionProc`. For the horizontal scroll bar, `DoContentClick` calls `TrackControl` and passes it a pointer to an action procedure called `MyHorzntlActionProc`. These action procedures are described in the next section.

Scrolling in Response to Events in Scroll Arrows and Gray Areas

With each click in a scroll arrow, your application should scroll by a distance of one unit (that is, by a single line, character, cell, or whatever your application deems appropriate) in the chosen direction. When the user holds the mouse button down while the cursor is in a scroll arrow, your application should scroll continuously by single units until the user releases the mouse button or until your application has scrolled as far as possible in the document.

With each click in a gray area, your application should scroll in the appropriate direction by a distance of just less than the height or width of one window to show part of the previous window (thus placing the newly displayed material in context). When the user holds the mouse button down while the cursor is in a gray area, your application should scroll continuously in units of this distance until the user releases the mouse button or until your application has scrolled as far as possible in the document.

When your application finishes scrolling, it should use `SetControlValue` to move the scroll box accordingly.

As previously described in this chapter, you use `FindControl` to determine when a mouse-down event has occurred in a control in one of your windows, and you use `TrackControl` to follow the movements of the cursor inside the control, to give the user visual feedback, and then to inform your application when the user releases the mouse button.

When a mouse-down event occurs in the scroll arrows or the gray areas of an active scroll bar, `FindControl` returns as its result the appropriate part code. The part codes for the scroll arrows and gray areas, and the constants you can use to represent them, are listed here:

Constant	Part code	Control part
`inUpButton`	20	Up scroll arrow for a vertical scroll bar, left scroll arrow for a horizontal scroll bar
`inDownButton`	21	Down scroll arrow for a vertical scroll bar, right scroll arrow for a horizontal scroll bar
`inPageUp`	22	Gray area above scroll box for a vertical scroll bar, gray area to left of scroll box for a horizontal scroll bar
`inPageDown`	23	Gray area below scroll box for a vertical scroll bar, gray area to right of scroll box for a horizontal scroll bar

When `FindControl` returns one of these part codes, your application should immediately call `TrackControl`. As long as the user holds down the mouse button while the cursor is in a scroll arrow, `TrackControl` highlights the scroll arrow, as shown in Figure 5-8 on page 5-12. When the user releases the mouse button, `TrackControl` removes the highlighting.

For all of the other standard controls, as well as for the scroll box in a scroll bar, your application doesn't respond until `TrackControl` reports a mouse-up event in the same control part where the mouse-down event initially occurred. However, for scroll arrows and gray areas, your application must respond by scrolling the document *before* `TrackControl` reports that the user has released the mouse button. When you call `TrackControl` for scroll arrows and gray areas, you must define an action procedure that scrolls appropriately until `TrackControl` reports that the user has released the mouse button.

When the user releases the mouse button or moves the cursor away from the scroll arrow or gray area, `TrackControl` returns as its result one of the previously listed values that represent the control part. As shown in Listing 5-18 on page 5-53, the `DoContentClick` procedure tests for the part codes `inUpButton`, `inDownButton`, `inPageUp`, and `inPageDown` to determine when a mouse-down event occurs in a scroll arrow or a gray area.

When the user presses or holds down the mouse button while the cursor is in either the scroll arrow or the gray area of the vertical scroll bar, `DoContentClick` calls `TrackControl` and passes it a pointer to an application-defined action procedure called `MyVerticalActionProc`. For the horizontal scroll bar, `DoContentClick` calls `TrackControl` and passes it a pointer to an action procedure called `MyVerticalActionProc`. In turn, `TrackControl` calls these action procedures to scroll continuously until the user releases the mouse button.

Note

As an alternative to passing a pointer to your action procedure in a parameter to `TrackControl`, you can use the `SetControlAction` procedure to store a pointer to the action procedure in the `contrlAction` field in the control record. When you pass `Pointer(-1)` instead of a procedure pointer to `TrackControl`, `TrackControl` uses the action procedure pointed to in the control record. ◆

Listing 5-19 shows two sample action procedures: `MyVerticalActionProc`—which responds to mouse events in the scroll arrows and gray areas of a vertical scroll bar—and `MyHorzntlActionProc`—which responds to those same events in a horizontal scroll bar. When `TrackControl` calls these action procedures, it passes a control handle and an integer representing the part of the control in which the mouse event occurred. Both `MyVerticalActionProc` and `MyHorzntlActionProc` use the constants `inUpButton`, `inDownButton`, `inPageUp`, and `inPageDown` to test for the control part passed by `TrackControl`.

Listing 5-19 Action procedures for scrolling through a text document

```
PROCEDURE MyVerticalActionProc (control: ControlHandle; part: Integer);
VAR
   scrollDistance:   Integer;
   window:           WindowPtr;
   myData:           MyDocRecHnd;
BEGIN
   IF part <> 0 THEN
   BEGIN
      window := control^^.contrlOwner; {get the control's window}
      myData := MyDocRecHnd(GetWRefCon(window));
      HLock(Handle(myData));
      CASE part OF
         inUpButton, inDownButton:  {get one line to scroll}
            scrollDistance := 1;
         inPageUp, inPageDown:   {get the window's height}
         BEGIN
            scrollDistance := (myData^^.editRec^^.viewRect.bottom -
                              myData^^.editRec^^.viewRect.top)
                              DIV myData^^.editRec^^.lineHeight;
               {subtract 1 line so user sees part of previous window}
            scrollDistance := scrollDistance - 1;
         END;
      END;  {of part CASE}
      IF (part = inDownButton) OR (part = inPageDown) THEN
         scrollDistance := -scrollDistance;
      MyMoveScrollBox(control, scrollDistance);
      IF scrollDistance <> 0 THEN   {scroll by line or by window}
         TEPinScroll(0, scrollDistance * myData^^.editRec^^.lineHeight,
                     myData^^.editRec);
      HUnLock(Handle(myData));
   END;
END; {of MyVerticalActionProc}

PROCEDURE MyHorzntlActionProc (control: ControlHandle; part: Integer);
VAR
   scrollDistance:   Integer;
   window:           WindowPtr;
   myData:           MyDocRecHnd;
BEGIN
   IF part <> 0 THEN
   BEGIN
      window := control^^.contrlOwner; {get the control's window}
      myData := MyDocRecHnd(GetWRefCon(window));
      HLock(Handle(myData));
      CASE part OF
         inUpButton, inDownButton:  {get a few pixels}
            scrollDistance := kButtonScroll;
         inPageUp, inPageDown:   {get a window's width}
            scrollDistance := myData^^.editRec^^.viewRect.right -
                               myData^^.editRec^^.viewRect.left;
      END;  {of part CASE}
      IF (part = inDownButton) OR (part = inPageDown) THEN
         scrollDistance := -scrollDistance;
      MyMoveScrollBox(control, scrollDistance);
      IF scrollDistance <> 0 THEN
         TEPinScroll(scrollDistance, 0, myData^^.editRec);
      HUnLock(Handle(myData));
   END;
END; {of MyHorzntlActionProc}
```

Each action procedure begins by determining an appropriate scrolling distance. For the scroll arrows in a vertical scroll bar, `MyVerticalActionProc` defines the scrolling distance as one line. For the gray areas in a vertical scroll bar, `MyVerticalActionProc` determines the scrolling distance in lines by dividing the window height by the line height; the window height is determined by subtracting the bottom coordinate of the view rectangle (defined in the edit record) from its top coordinate. Then `MyVerticalActionProc` subtracts 1 from this distance so that when the user presses the mouse button while the cursor is in a gray area, `MyVerticalActionProc` scrolls one line less than the total number of lines in the window.

The `MyVerticalActionProc` procedure later multiplies these line distances by the line height to derive pixel distances to pass in parameters to `TEPinScroll`. Also, `MyVerticalActionProc` turns these distances into negative values when the mouse-down event occurs in the lower scroll arrow or in the gray area below the scroll box.

For the scrolling distance of the scroll arrows in horizontal scroll bars, `MyHorzntlActionProc` uses a predetermined pixel distance—roughly the document's average character width. For the scrolling distance of the gray areas `MyHorzntlActionProc` uses the window width (which is derived by subtracting the left coordinate of the view rectangle from its right coordinate). The `MyHorzntlActionProc` routine turns these distances into negative values when the mouse-down event occurs in the right scroll arrow or in the gray area to the right of the scroll box.

After calling `MyMoveScrollBox`, an application-defined routine that moves the scroll box, both action procedures use `TEPinScroll` to move the text displayed in the window by the scrolling distance. (In this example, the SurfWriter application is scrolling a simple monostyled text document stored as a TextEdit edit record. For a discussion of using the more general-purpose QuickDraw scrolling routine `ScrollRect`, see the previous section, "Scrolling in Response to Events in the Scroll Box" beginning on page 5-53.)

The `TEPinScroll` procedure automatically creates an update region and invokes an update event. In its window-updating code, the SurfWriter application uses the `TEUpdate` procedure to draw the text in the update region, as shown in Listing 5-23 on page 5-65.

The action procedures continue moving the text by the specified distances over and over until the user releases the mouse button and `TrackControl` completes. If there is no more area to scroll through, `TEPinScroll` automatically stops scrolling, as your application should if you implement your own scrolling routine.

Listing 5-20 shows how the application-defined procedure `MyMoveScrollBox` uses `GetControlValue`, `GetControlMaximum`, and `SetControlValue` to move the scroll box an appropriate distance while the action procedures scroll through the document. The `MyMoveScrollBox` procedure uses `GetControlMaximum` to determine the maximum scrolling distance, `GetControlValue` to determine the current setting for the scroll box, and `SetControlValue` to assign the new setting and move the scroll box. Use of the `SetControlMaximum` and `SetControlValue` routines is described in "Determining and Changing Control Settings" beginning on page 5-37; `GetControlMaximum` is described in detail on page 5-104.

Listing 5-20 Moving the scroll box from the action procedures

```
PROCEDURE MyMoveScrollBox (control: ControlHandle;
                           scrollDistance: Integer);
VAR
   oldSetting, setting, max:  Integer;
BEGIN
   oldSetting := GetControlValue(control);   {get last setting}
   max := GetControlMaximum(control);        {get maximum setting}
{subtract action procs' scroll amount from last setting to get new setting}
   setting := oldSetting - scrollDistance;
   IF setting < 0 THEN
      setting := 0
   ELSE IF setting > max THEN
      setting := max;
   SetControlValue(control, setting); {assign new current setting}
END; {of MyMoveScrollBox}
```

The previous two sections have described how to move the bits displayed in the window; the next section describes how to draw into the update region.

Drawing a Scrolled Document Inside a Window

The previous two sections have described how to use the QuickDraw procedure `ScrollRect` and the TextEdit procedure `TEPinScroll` in response to the user manipulating any of the five parts of a scroll bar. After using these or your own routines for moving the bits in your window, your application must draw into the update region. Typically, you use your own window-updating code for this purpose.

Both `InvalRect` and `TEPinScroll`, which are used in the examples shown earlier in this chapter, create update regions that cause update events. As described in the chapters “Window Manager” and “Event Manager” in this book, your application should draw in the update regions of your windows when it receives update events. If you create your own scrolling routine to use instead of `ScrollRect` or `TEPinScroll`, you should guarantee that it generates an update event or that it explicitly calls your own window-updating routine.

Listing 5-21 shows an application-defined routine, `DoUpdate`, that the SurfWriter application calls whenever it receives an update event. In this procedure, the application tests for two different types of windows: windows containing graphics objects and windows containing text created with TextEdit routines.

Listing 5-21 An application-defined update routine

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
   windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyGraphicsWindow:   {window containing graphics objects}
      BEGIN
         BeginUpdate(window);
         MyDrawGraphicsWindow(window);
         EndUpdate(window);
      END;  {of updating graphics windows}
   kMyDocWindow:        {window containing TextEdit text}
      BEGIN
         BeginUpdate(window);
         MyDrawWindow(window);
         EndUpdate(window);
      END;  {of updating TextEdit document windows}
   {handle other window types——modeless dialogs, etc.——here}
   END;  {of windowType CASE}
END;  {of DoUpdate}
```

In this example, when the window requiring updating is of type kMyGraphicsWindow, DoUpdate uses another application-defined routine called MyDrawGraphicsWindow. When the window requiring updating is of type kMyDocWindow, DoUpdate uses another application-defined routine—namely, MyDrawWindow. Listing 5-22 shows the MyDrawGraphicsWindow routine and Listing 5-23 on page 5-65 shows the MyDrawWindow routine.

Before drawing into the scrolled-away portion of the window, both of these routines use the QuickDraw, Window Manager, and Control Manager routines necessary for updating windows. ("Updating a Control" beginning on page 5-29 describes the UpdateControls procedure; see the chapter "Window Manager" in this book for a detailed description of how to use the rest of these routines to update a window.)

Listing 5-22 Redrawing a window containing graphics objects

```
PROCEDURE MyDrawGraphicsWindow (window: WindowPtr);
VAR
   myData:  MyDocRecHnd;
   i:       Integer;
BEGIN
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
      BEGIN
         EraseRect(portRect);
         UpdateControls(window, visRgn);
         DrawGrowIcon(window);
         SetOrigin(GetControlValue(myData^^.hScrollBar),
                   GetControlValue(myData^^.vScrollBar));
         i := 1;
         WHILE i <= myData^^.numObjects DO
            DrawMyObjects(portRect, myData^^.numObjects[i]);
            i := i + 1;
         END; {of WHILE}
         SetOrigin(0, 0);
      END;
   HUnLock(Handle(myData));
END;  {of MyDrawGraphicsWindow}
```

The MyDrawGraphicsWindow routine uses the QuickDraw procedure SetOrigin to change the window origin by an amount equal to the scroll bar settings, so that the upper-left corner of the document lies at (0,0) in the window's local coordinate system. The SurfWriter sample application performs this step so that its own drawing routines can draw into the correct area of the window.

Notice that MyDrawGraphicsWindow calls SetOrigin only after calling the necessary Window Manager and Control Manager routines, because the Window Manager and Control Manager always expect the window origin to be at (0,0).

By using SetOrigin to change the window origin, MyDrawGraphicsWindow can treat the objects in its document as being located in a coordinate system where the upper-left corner of the document is always at (0,0). Then MyDrawGraphicsWindow calls another of its own routines, DrawMyObjects, to draw the objects it has stored in its document record for the window.

After performing all its own drawing in the window, MyDrawGraphicsWindow again uses SetOrigin—this time to reset the window origin to (0,0) so that future Window Manager and Control Manager routines will draw into the correct area of the window.

Figure 5-16 through Figure 5-20 earlier in this chapter help to illustrate how to use SetOrigin to offset the window's coordinate system so that you can treat the objects in your document as fixed in the document's own coordinate space. However, it is not necessary for your application to use SetOrigin. Your application can leave the window's coordinate system fixed and instead offset the items in your document by the amount equal to the scroll bar settings. The QuickDraw procedures OffsetRect, OffsetRgn, SubPt, and AddPt, which are described in *Inside Macintosh: Imaging,* are useful if you pursue this approach.

Note
The SetOrigin procedure does not move the window's clipping region. If you use clipping regions in your windows, use the QuickDraw procedure GetClip to store your clipping region immediately after your first call to SetOrigin. Before calling your own window-drawing routine, use the QuickDraw procedure ClipRect to define a new clipping region—to avoid drawing over your scroll bars, for example. After calling your own window-drawing routine, use the QuickDraw procedure ClipRect to restore the original clipping region. You can then call SetOrigin again to restore the window origin to (0,0) with your original clipping region intact. See *Inside Macintosh: Imaging* for detailed descriptions of clipping regions and of these QuickDraw routines. ◆

The previous examples in this chapter have shown an application that uses a TextEdit edit record to store the information created by the user. For simple text-handling needs, TextEdit provides many routines that simplify your work; for example, the TEPinScroll procedure (used in Listing 5-18 on page 5-53 and Listing 5-19 on page 5-59) resets the view rectangle of text stored in an edit record by the amount of pixels specified by the application. The TEPinScroll procedure then generates an update event for the window. The TextEdit procedure TEUpdate should then be called in an application's update routine to draw the update region of the scrolled window.

Listing 5-23 shows an application-defined procedure, MyDrawWindow, that uses TEUpdate to update the text in windows of type kMyDocWindow. The TEUpdate procedure manages all necessary shifting of coordinates during window updating, so MyDrawWindow does not have to call SetOrigin as it does when it uses ScrollRect.

Listing 5-23 Redrawing a window after scrolling a TextEdit edit record

```
PROCEDURE MyDrawWindow (window: WindowPtr);
VAR
   myData: MyDocRecHnd;
BEGIN
   SetPort(window);
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^ DO
      BEGIN
         EraseRect(portRect);
         UpdateControls(window, visRgn);
         DrawGrowIcon(window);
         TEUpdate(portRect, myData^^.editRec);
      END;
   HUnLock(Handle(myData));
END;  {of MyDrawWindow}
```

Moving and Resizing Scroll Bars

As described earlier in “Creating Scroll Bars” beginning on page 5-21, your application initially defines the location of a scroll bar within a window—and the size of the scroll bar—by specifying a rectangle in a control resource or in a parameter to `NewControl`. However, your application must be able to size and move the scroll bar dynamically in response to the user’s resizing of your windows.

The chapter “Window Manager” in this book describes how to size windows when your application opens them and how to resize them—for example, in response to the user dragging the size box or clicking the zoom box. This section describes how to move and resize your scroll bars so that they fit properly on the right and bottom edges of your windows.

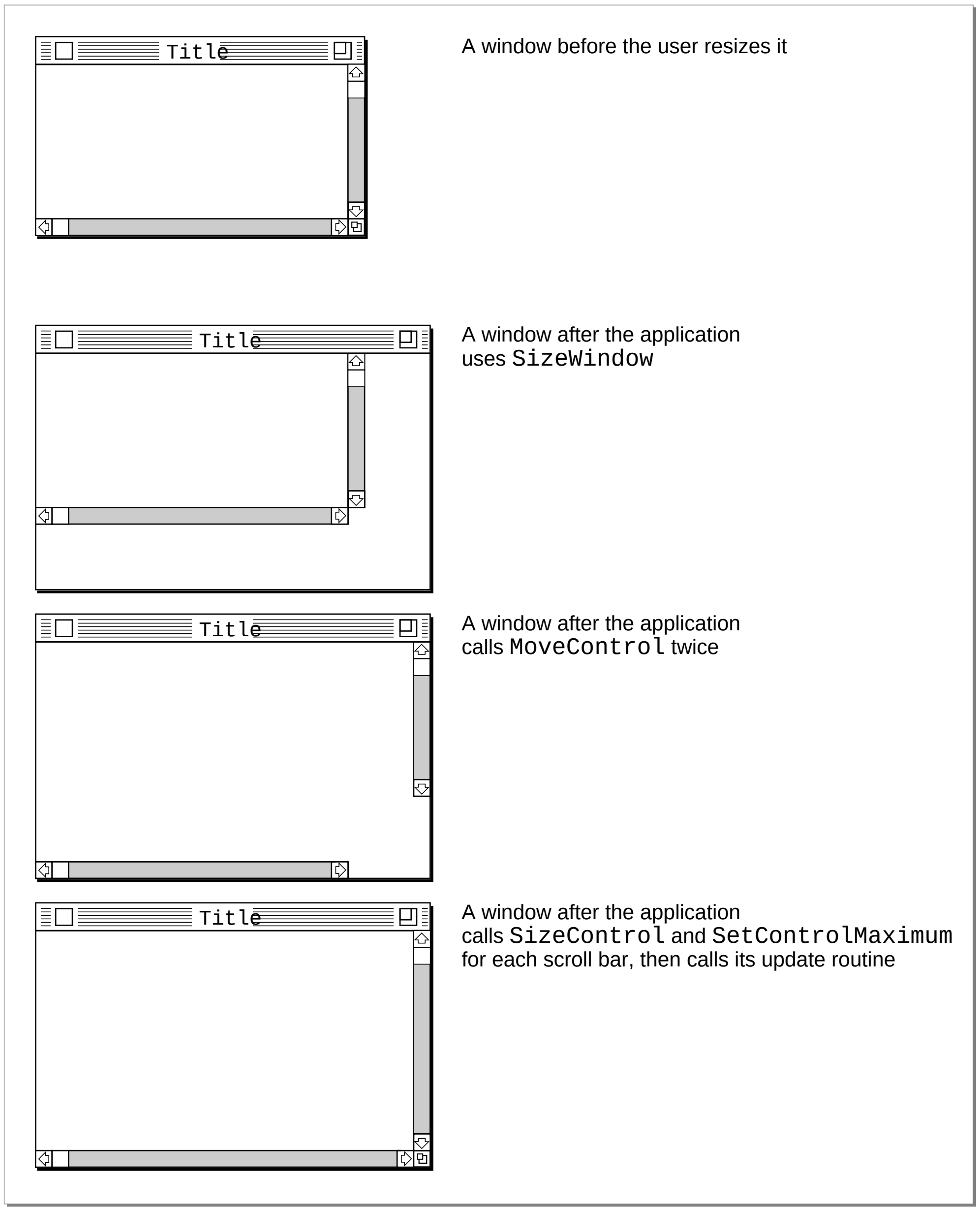
When resizing your windows, your application should perform the following steps to adjust each scroll bar.

1. Resize the window.
2. Use the `HideControl` procedure to make each scroll bar invisible.
3. Use the `MoveControl` procedure to move the vertical scroll bar to the right edge of the window, and use the `MoveControl` procedure to move the horizontal scroll bar to the bottom edge of the window.
4. Use the `SizeControl` procedure to lengthen or shorten each scroll bar, so that each extends to the size box in the lower-right corner of the window.
5. Recalculate the maximum settings for the scroll bars and use `SetControlMaximum` to update the settings and to redraw the scroll boxes appropriately. (Remember, you derive a scroll bar’s maximum setting by subtracting the length or width of its window from the length or width of the document.)

6. Use the `ShowControl` procedure to make each scroll bar visible in its new location.

Figure 5-21 illustrates how to move and resize scroll bars in a resized window; if your application neglected to use the `HideControl` procedure, the user would see each of these steps as it took place.

Figure 5-21 Moving and resizing scroll bars


Title
A window before the user resizes it
Title
A window after the application uses SizeWindow
Title
A window after the application calls MoveControl twice
Title
A window after the application calls SizeControl and SetControlMaximum for each scroll bar, then calls its update routine

Listing 5-14 on page 5-39 shows an application-defined routine, `MyAdjustScrollBars`, that is called when the user opens a new window, opens an existing document in a window, or resizes a window.

When it creates a window, `MyAdjustScrollBars` stores handles to each scroll bar in a document record. By dereferencing the proper fields of the document record, `MyAdjustScrollBars` passes handles for the vertical and horizontal scroll bars to the `HideControl` procedure, which makes the scroll bars invisible. By making the scroll bars invisible until it has finished manipulating them, `MyAdjustScrollBars` ensures that the user won't see the scroll bars blinking in different locations onscreen.

When `MyAdjustScrollBars` needs to adjust the size or location of either of the scroll bars, it calls another application-defined routine, `MyAdjustScrollSizes`, which is shown in Listing 5-24.

Listing 5-24 Changing the size and location of a window's scroll bars

```
CONST
   kScrollbarWidth = 16;                     {conventional width}
   kScrollbarAdjust = kScrollbarWidth - 1;   {to align with window frame}
   kScrollTweek = 2;                   {to align scroll bars with size box}

PROCEDURE MyAdjustScrollSizes (window: WindowPtr);
VAR
   teRect:                          Rect;
   myData:                          MyDocRecHnd;
   teTop, teRight, teBottom,teLeft: Integer;
BEGIN
   MyGetTERect(window, teRect);  {calculate the teRect based on the }
                                 { portRect, adjusted for the scroll bars}
   myData := MyDocRecHnd(GetWRefCon(window));
   HLock(Handle(myData));
   WITH window^.portRect DO
   BEGIN
      teTop := top;
      teRight := right;
      teBottom := bottom;
      teLeft := left;
   END;
   WITH myData^^ DO
   BEGIN
      editRec^^.viewRect := teRect;    {set the viewRect}
      MyAdjustViewRect(editRec);       {snap to nearest line}
      {move the controls to match the new window size}
      MoveControl(vScrollBar, teRight - kScrollbarAdjust, -1);
      SizeControl(vScrollBar, kScrollbarWidth, (teBottom - teTop) -
                 (kScrollbarAdjust - kScrollTweek));
      MoveControl(hScrollBar, -1, teBottom - kScrollbarAdjust);
      SizeControl(hScrollBar, (teRight - teLeft) -
            (kScrollbarAdjust - kScrollTweek), kScrollbarWidth);
   END;
   HUnLock(Handle(myData));
END; {of MyAdjustScrollSizes}
```

The `MyAdjustScrollSizes` routine uses the boundary rectangle of the window's content region—which is stored in the `portRect` field of the window record—to determine the size of the window. To move the scroll bars to the edges of the window, `MyAdjustScrollSizes` uses the `MoveControl` procedure.

The `MoveControl` procedure takes three parameters: a handle to the control being moved, the horizontal coordinate (local to the control's window) for the new location of the upper-left corner of the control's rectangle, and the vertical coordinate for that new location. The `MoveControl` procedure moves the control to this new location and changes the rectangle specified in the `controlRect` field of the control's control record.

In Listing 5-24, `MyAdjustScrollSizes` passes to `MoveControl` the handles to the scroll bars. (The SurfWriter sample application stores the handle in its document record for the window.)

Figure 5-22 illustrates the location of a vertical scroll bar before it is moved to a new location within its resized window.

To determine a new horizontal (that is, left) coordinate of the upper-left corner of the vertical scroll bar, `MyAdjustScrollSizes` subtracts 15 from the right coordinate of the window. As shown in Figure 5-23, this puts the right edge of the 16-pixel-wide scroll bar directly over the 1-pixel-wide window frame on the right side of the window.

In Listing 5-24 on page 5-67, `MyAdjustScrollSizes` specifies –1 as the vertical (that is, top) coordinate of the upper-left corner of the vertical scroll bar. As shown in Figure 5-23, this places the top edge of the scroll bar directly over the 1-pixel-wide line at the bottom of the title bar. (The bottom line of the title bar has a vertical value of –1 in the window's local coordinate system.)

The `MyAdjustScrollSizes` routine specifies –1 as the horizontal coordinate of the upper-left corner of the horizontal scroll bar; this puts the left edge of the horizontal scroll bar directly over the 1-pixel-wide window frame. (The left edge of the window frame has a horizontal value of –1 in the window's local coordinate system.)

To fit your scroll bars inside the window frame properly, you should set the top coordinate of a vertical scroll bar at –1 and the left coordinate of a horizontal scroll bar at –1, unless your application uses part of the window's scroll regions opposite the size box for displaying information or additional controls. For example, you may choose to display the current page number of the document in the lower-left corner of a window. In this case, specify a left coordinate so that the horizontal scroll bar doesn't obscure this area.

Figure 5-22 A vertical scroll bar before the application moves it within a resized window

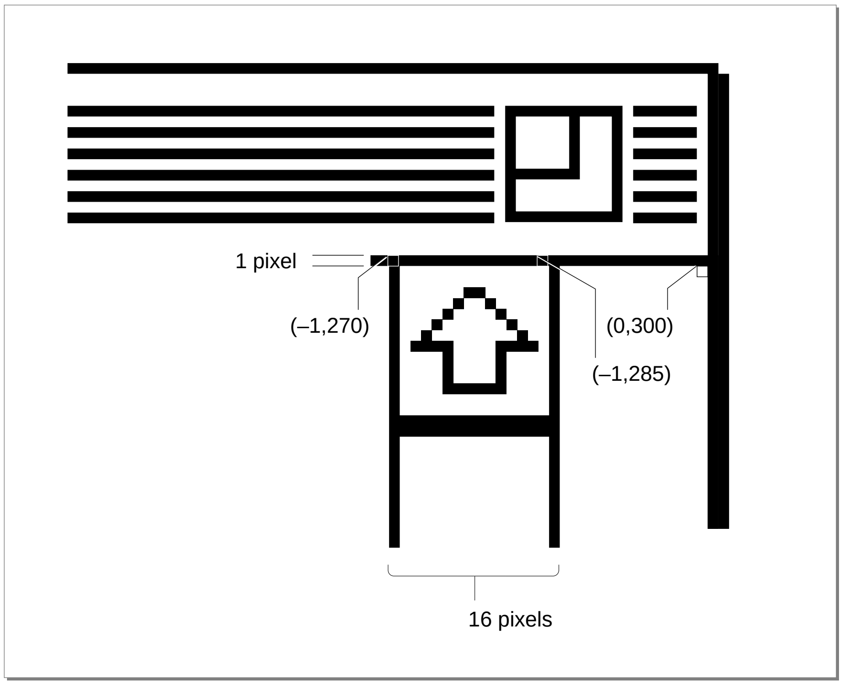
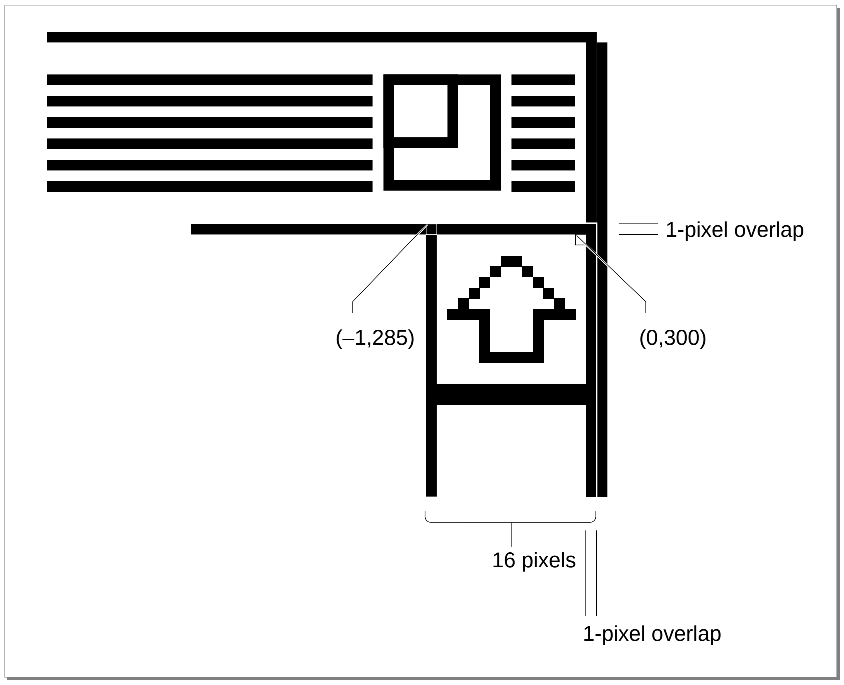


Figure 5-23 A vertical scroll bar after the application moves its upper-left point



See *Macintosh Human Interface Guidelines* for a discussion of appropriate uses of a window's scroll areas for items other than scroll bars.

To determine a new vertical coordinate for the upper-left corner of the horizontal scroll bar, `MyAdjustScrollSizes` subtracts 15 from the bottom coordinate of the window; this puts the bottom edge of the scroll bar directly over the window frame at the bottom of the window.

The `MoveControl` procedure moves the upper-left corner of a scroll bar so that it's in the proper location within its window frame. To make the vertical scroll bar fit the height of the window, and to make the horizontal scroll bar fit the width of the window, `MyAdjustScrollSizes` then uses the `SizeControl` procedure.

The `SizeControl` procedure takes three parameters: a handle to the control being sized, a width in pixels for the control, and a height in pixels for the control. When resizing a vertical scroll bar, you adjust its height; when resizing a horizontal scroll bar, you adjust its width.

When using `SizeControl` to adjust the vertical scroll bar, `MyAdjustScrollSizes` passes a constant representing 16 pixels for the vertical scroll bar's width, which is the conventional size.

To determine the proper height for this scroll bar, `MyAdjustScrollSizes` first derives the height of the window by subtracting the top coordinate of the window's rectangle from its bottom coordinate. Then `MyAdjustScrollSizes` subtracts 13 pixels from this window height and passes the result to `SizeControl` as the height of the vertical scroll bar. The `MyAdjustScrollSizes` routine subtracts 13 pixels from the window height to leave room for the 16-pixel-high size box (at the bottom of the window) minus three 1-pixel overlaps: one at the top of the window frame, one at the top of the size box, and one at the bottom of the size box.

When using `SizeControl` to adjust the horizontal scroll bar, `MyAdjustScrollSizes` passes a constant representing 16 pixels—the conventional height of the horizontal scroll bar. To determine the proper width of this scroll bar, `MyAdjustScrollSizes` first derives the width of the window by subtracting the left coordinate of the window's rectangle from its right coordinate. From this window width, `MyAdjustScrollSizes` then subtracts 13 pixels to allow for the size box (just as it does when determining the height of the vertical scroll bar).

When `MyAdjustScrollSizes` completes, it returns to `MyAdjustScrollBars`, which then uses another of its own routines, `MyAdjustScrollValues`. In turn, `MyAdjustScrollValues` calls `MyAdjustHV` (shown in Listing 5-16 on page 5-41), which recalculates the maximum settings for the scroll bars and uses `SetControlMaximum` to update the maximum settings and redraw the scroll boxes appropriately.

When `MyAdjustHV` completes, it eventually returns to the SurfWriter application's `MyAdjustScrollBars` procedure, which then uses the `ShowControl` procedure to make the newly adjusted scroll bars visible again.

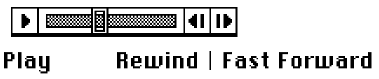
Defining Your Own Control Definition Function

The Control Manager allows you to implement controls other than the standard ones (buttons, checkboxes, radio buttons, pop-up menus, and scroll bars). To implement nonstandard controls, you must define your own control definition functions. Typically, the only types of controls you might need to implement are sliders or dials, which are similar to scroll bars in that they graphically represent a range of values the user can set. As scroll bars have scroll boxes, your sliders and dials should have indicators for setting values and indicating current settings.

Dials and sliders display the value, magnitude, or position of something, typically in some pseudo-analog form—for instance, the position of a sliding switch, the reading on a scale, or the angle of a needle on a gauge; the setting may be displayed digitally as well. The user should be able to change the control's setting by dragging its indicator.

Figure 5-24 illustrates a control supported by an application-defined control definition function. This control might be used to play back a sound or a QuickTime movie. The application might wish to define the control so that it plays the sound or movie at normal speed when the user clicks the control part on the left. The application might use the indicator along the slider to show what portion of the entire sound or movie sequence is currently playing. The application also allows the user to move quickly forward and backward through the sequence by dragging the indicator. Finally, the application might wish to define the two control parts on the far right so that they play backward (that is, "rewind") and play forward quickly (that is, "fast forward"), respectively, when the user clicks them.

Figure 5-24 A custom control



Note
When you design a dial or slider, be sure to include meaningful labels that indicate to users the range and the direction of the indicator. ◆

Rather than create such a control yourself, you might be tempted to use a scroll bar for this purpose. Do not do so. Using a scroll bar for any purpose other than scrolling through a window compromises the consistency of the Macintosh interface.

To define your own nonstandard control, you must write a control definition function, compile it as a resource of type `'CDEF'`, and include it in your resource file. (For more information about creating resources, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.)

When you use Control Manager routines, they in turn call your control definition function as necessary. For example, for the control in Figure 5-24 to work properly, its control definition function must be able to

- draw the control—including repositioning its indicator, making it inactive or active, and highlighting its control parts appropriately when mouse events occur in them
- determine when a mouse-down event occurs in a control part
- calculate the region of the control and its indicator
- move the indicator and update the control record with a new setting

You can also use your control definition function to modify or expand certain Control Manager behaviors; for example, you can implement your own manner of dragging an indicator, and you can perform your own type of control initialization.

For details about writing a control definition function, see "Defining Your Own Control Definition Function" beginning on page 5-109.

Control Manager Reference

This section describes the data structures, routines, and resources that are specific to the Control Manager.

The "Data Structures" section shows the data structures for the control record, the auxiliary control record, the pop-up menu private data record, and the control color table record. The "Control Manager Routines" section describes Control Manager routines for creating controls, drawing controls, handling mouse events in controls, changing control settings and display, determining control settings, and removing controls. The "Application-Defined Routines" section describes the control definition function, which you need to provide when defining your own controls. The "Application-Defined Routines" section also describes the action procedure, which defines an action to be performed repeatedly as long as the user holds down the mouse button while the cursor is in a control. The "Resources" section describes the control resource and the control color table resource.

Data Structures

This section describes the control record, the auxiliary control record, the pop-up menu private data record, and the control color table record.

Your application doesn't specifically create the control record, the auxiliary control record, or the pop-up menu private data record; rather, your application simply creates any necessary resources and uses the appropriate Control Manager routines. The Control Manager creates these records as necessary.