5 Version 1.1

CONTROLS

Includes Demonstration Programs Controls1 & Controls2

Introduction

Controls are on-screen objects which the user can manipulate to cause an immediate action or to change settings to modify a future action.

You can use the Control Manager to create and manage controls. An alternative method is to use the Dialog Manager to more easily create and manage controls in alert boxes or dialog boxes. Note, however, that the Control Manager is usually used to implement the more complex dialog boxes.

Every control you create must be associated with a particular window. All the controls for a window are stored in a **control list** referenced by the window's window record.

Standard and Other Controls

The **standard controls** provided by the Control Manager are illustrated at Fig 1.

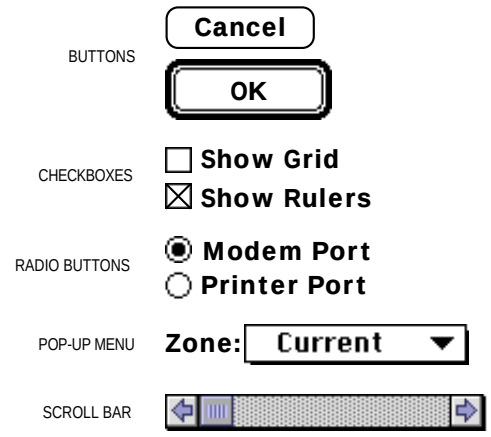

FIG 1 - STANDARD CONTROLS PROVIDED BY THE CONTROL MANAGER

The Control Manager displays these controls in colours which provide consistency across all monitors, from black and white to colour displays. To retain this consistency, you should not change the default colours.

Buttons

You normally use buttons in alert boxes and dialog boxes. Buttons typically allow the user to perform actions instantaneously, for example, completing the actions in a dialog box or acknowledging an error in an alert box. 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. (In alert boxes, the Dialog Manager automatically outlines the default button; however, your application must outline the button in dialog boxes.)

Your application should respond to key-down events involving the Enter and Return keys as if the user had clicked the default button.

Checkboxes

Checkboxes are typically used in dialog boxes so that the user can supply additional information necessary for completing a command. Checkboxes provide alternative choices and act like toggle switches, turning a setting on or off. `SetControlValue` is used to place an X in the box when the user selects it and to remove the X when the user deselects it.

Each checkbox has a title, which should reflect two clearly opposite states. If you cannot devise a title which clearly implies two opposite states, you might be better off providing two radio buttons.

Radio Buttons

Like checkboxes, radio buttons retain and display an on or off setting and are typically used inside dialog boxes. Radio buttons represent choices that are related but not necessarily opposite. `SetControlValue` is used to fill a selected 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 within a group are mutually exclusive. The Control Manager, however, cannot tell 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 group of radio buttons must comprise at least two radio buttons. Each group must have a label which identifies the kind of choices the group offers and each button must have a title identifying what the radio button does. 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.

Pop-Up Menus

Pop-up menus provide the user with a simple way to choose from 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 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 group of radio buttons, the items in a pop-up menu are mutually exclusive.

Scroll Bars

Scroll bars change the portion of a document that the user can view within a 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 box is called the **gray area**. If the user drags the scroll box, clicks a scroll arrow or clicks in the gray region, your application scrolls the document accordingly.

Scroll Box

`SetControlValue` or `SetControlMaximum` are used 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. If the user drags the scroll box, the Control Manager redraws the scroll box in its new position. You then use `GetControlValue` to determine the position of the control box, and to display the appropriate portion of the document.

Scroll Arrows

When the scroll arrows are clicked, your application uses `SetControlValue` to move the scroll box in the direction of the arrow being clicked. Each click should move the document one unit in the chosen direction. (In a text document, a unit would typically be one line of text.)

Gray Area

When the gray area is clicked above the scroll box, your application should move the document up so that the bottom line of the previous view is at the top of the new view, and it should move the scroll box accordingly. A similar, but downward movement, should occur when the user clicks in the gray area below the scroll box.

Custom Controls

If you need controls other than the standard controls, you can design and implement your own **custom controls**. Typically, the only types of controls you might need to implement are **sliders** or **dials** to represent a range of values[1].

If you need a custom control, you must provide your own control definition function. Custom control definition functions are addressed at Chapter 19 — Custom Control Definition Functions and VBL Tasks.

Visual Feedback From Controls

The `TrackControl` function, which is called in response to a mouse-down event in a control, provides visual feedback when a mouse-down occurs in an active control by:

- Displaying buttons in inverse video.
- Drawing checkboxes and radio buttons with heavy lines.
- Highlighting the titles of, and displaying the items in, pop-up menus.
- Highlighting the scroll arrows.
- Moving outlines of scroll boxes when the user drags them.

Active and Inactive Controls

A control can be either **active** or **inactive**. Whenever it is inappropriate for your application to respond to a mouse-down event in a control, you should make it inactive. The Control Manager continues to display inactive controls so that they remain visible, but in a manner which indicates their state to the user. The Control Manager:

- Dims inactive buttons, checkboxes, radio buttons and pop-up menus. (Actually, only the titles of buttons, checkboxes and radio buttons are dimmed.)
- Lightens the gray area and removes the scroll box from inactive scroll bars.

Activating and Deactivating Controls Other Than Scroll Bars

You use `HiliteControl` to make active buttons, checkboxes, radio buttons and pop-up menus inactive and vice versa. You should make buttons, checkboxes, radio buttons and pop-up menus inactive when they are not relevant to the current context and when their windows are not frontmost.

---

[1] A scroll bar is a slider representing the entire contents of a document, and the user uses the scroll box to move to a specific location in that document. To conform to user interface guidelines, do not use scroll bars to represent any other concept (for example, for changing a setting).

Activating and Deactivating Scroll Bars

You make scroll bars inactive when the document is smaller than the window in which it is being displayed. To make a scroll bar inactive, you typically use `SetControlMaximum` to make the scroll bar's maximum value equal to its minimum value, which causes the Control Manager to automatically make the scroll bar inactive and display it in the inactive state. To make the scroll bar active again, `SetControlMaximum` should be used to set its maximum value larger than its minimum value.

Hiding and Showing Controls

`HideControl` should be used to hide scroll bars when their windows are not frontmost. `HideControl` erases a control by filling its enclosing rectangle with the owning window's background pattern. `ShowControl` reverses this situation. Hiding a control is not the same as making a control inactive.

The Control Definition Function

A **control definition function** determines the appearance and behaviour of a control. Various Control Manager routines call a control definition function when they need to perform some control-related action.

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 ID 0 defines the appearance and behaviour of buttons, checkboxes and radio buttons.
- The `'CDEF'` resource with ID 1 defines the appearance and behaviour of scroll bars.
- The `'CDEF'` resource with ID 63 defines the appearance and behaviour of pop-up menus.

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 control. You specify a particular control with a **control definition ID**, which is an integer containing the resource ID of the control definition function in the upper 12 bits and the variation code in the lower four bits.

The control definition ID is arrived at by multiplying the resource ID by 16 and adding the variation code. The following shows the control definition IDs for the standard controls, together with the derivation of those IDs:

Control	Resource ID	Variation Code	Control Definition ID (Decimal)	Control Definition ID (Constant)
Button	0	0	`0 * 16 + 0 = 0`	`pushButProc`
Checkbox	0	1	`0 * 16 + 0 = 1`	`checkBoxProc`
Radio button	0	2	`0 * 16 + 0 = 2`	`radioButProc`
Scroll bar	1	0	`1 * 16 + 0 = 16`	`scrollBarProc`
Pop-up menu	63	0	`63 * 16 + 0 = 1008`	`popupMenuProc`

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

Creating and Displaying Controls

Creating a `'CNTL'` Resource

The first step in creating a control is to create a `'CNTL'` resource. An example of a `'CNTL'` resource, in Rez input format, is as follows:

```
resource 'CNTL' (rCancelButton, preload, purgeable)
{
  {87, 187, 107, 247},  /* Rectangle (local coordinates) for size and location. */
  0,                    /* Initial setting of control. */
  visible,              /* Make control visible. */
  1,                    /* Maximum setting of control. */
  0,                    /* Minimum setting of control. */
  pushButProc,          /* Control Definition ID. */
  0,                    /* Reference constant for application use. */
  "Cancel"              /* Title of control. */
};
```

Rectangle. The example resource is for a button. Buttons are drawn to fit the specified 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. For a checkbox or radio button, 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.

Initial, Minimum and Maximum Settings. For buttons, checkboxes and radio buttons, these settings should be supplied in the initial, maximum, and minimum setting fields:

- For buttons, which do not retain a setting, specify 0 for the initial and minimum settings field and 1 in the maximum settings field.

- For checkboxes and radio buttons, which retain an on-off setting, specify 0 when you want the control to be initially off. To turn a checkbox or radio button on, assign an initial setting of 1, which will cause the Control Manager to place an X in a checkbox or a black dot in a radio button. The maximum and minimum settings should be specified as 1 and 0 respectively.

Control Definition ID. The example specifies a button in the control definition ID field. For checkboxes, specify `checkBoxProc` and for radio boxes, specify `radioButProc`. Add the constant `popupUseWFont` to cause the control's text to be drawn in the current graphics port's font rather than the system font.

Reference Constant. Except when you add the `popupUseAddResMenu` variation code to the `popupMenuProc` control definition ID (see below), the reference constant field may be used for any purpose.

Title. The title of the control[2] is specified at the last field. By default, the Control Manager displays the title in the system font. When specifying a title, make sure that it will fit into the control's rectangle, otherwise the Control Manager will truncate the title.[3] For scroll bars, the title field should contain an empty string.)

Note that the values you supply in a control resource for a pop-up menu differ from those you specify for buttons, checkboxes, radio buttons and scroll bars. (See below.)

A further example `'CNTL'` resource, this time for a group of three radio buttons, is as follows:

```
resource 'CNTL' (cDroplet, preload, purgeable)
{
  {13, 23, 31, 142},   /* Rectangle (local coordinates) for size and location. */
  1,                   /* Initial setting of control. */
  visible,             /* Make control visible. */
  1,                   /* Maximum setting of control. */
  0,                   /* Minimum setting of control. */
  radioButProc,        /* Control Definition ID. */
  0,                   /* Reference constant for application use. */
  "Droplet"            /* Title of control. */
};
```

---

[2]Book title style should be used, ie, capitalize one word titles, nouns, adjectives, verbs and prepositions of four or more letters in multiple word titles.

[3]The Control Manager allows button, checkbox and radio button titles on multiple lines. End each line with the character code `0x0D` (carriage return). If the control is a button, each line is horizontally centered.

```
resource 'CNTL' (cQuack, preload, purgeable)
{
  {31, 23, 49, 142},   /* Rectangle (local coordinates) for size and location. */
  0,                   /* Initial setting of control. */
  visible, 1, 0, radioButProc, 0, "Quack"};
};

resource 'CNTL' (cWildEep, preload, purgeable)
{
  {49, 23, 67, 142},   /* Rectangle (local coordinates) for size and location. */
  0,                   /* Initial setting of control. */
  visible, 1, 0, radioButProc, 0, "WildEep"};
};
```

Creating a Control

`GetNewControl` and `NewControl` are used to create a new control in a window. You usually use `GetNewControl`, which takes a `'CNTL'` resource ID and a pointer to the window, creates a data structure called a **control record** from the information in the resource, adds the control record to the control list for your window, and returns a handle to the control. A control record is defined by the data type `ControlRecord`:

```
struct ControlRecord
{
  ControlRef        nextControl;    // Next Control.
  WindowRef         contrlOwner;    // Control's window.
  Rect              contrlRect;     // Rectangle.
  UInt8             contrlVis;      // 255 if visible, else 0.
  UInt8             contrlHilite;   // Highlight state.
  SInt16            contrlValue;    // Current setting.
  SInt16            contrlMin;      // Minimum Setting.
  SInt16            contrlMax;      // Maximum setting.
  Handle            contrlDefProc;  // Control definition function.
  Handle            contrlData;     // Data used by contrlDefProc.
  ControlActionUPP  contrlAction;   // Action procedure.
  SInt32            contrlRfCon;    // Reference constant.
  Str255            contrlTitle;    // Title.
};

typedef struct ControlRecord ControlRecord;
typedef ControlRecord *ControlPtr, **ControlHandle;

typedef ControlHandle ControlRef;
```

If the `'CNTL'` resource specifies that a control is initially visible, the Control Manager uses the control definition function to draw the control. (The Control Manager draws the control immediately and does not wait for the window updating mechanism.) If the `'CNTL'` resource specifies that the control is to be initially invisible, `ShowControl` may be used to draw the control when required.

Note that when you use the Dialog Manager to implement buttons, radio buttons, checkboxes or pop-up menus in alert boxes or dialog boxes, Dialog Manager routines automatically use Control Manager routines to create the controls for you.

Updating Controls

When your application receives an update event for a window containing controls, your application should call `UpdateControls` between the `BeginUpdate` and `EndUpdate` calls in its updating code.

Note that when you use the Dialog Manager to implement buttons, radio buttons, checkboxes or pop-up menus in alert boxes or dialog boxes, Dialog Manager routines automatically use Control Manager routines to update the controls for you.

Removing Controls

When you no longer need a control in a window that you wish to keep, you use `DisposeControl` to remove it from the screen, delete it from the window's control list, and release the control record and associated data structures from memory. `KillControls` will dispose of all of a window's controls at once.

Creating Scroll Bars

The `'CNTL'` resource for scroll bars should specify `scrollBarProc` as the control definition ID. Typically, you make the scroll bar invisible, set the initial, minimum and maximum settings to 0 and supply an empty string for the title.

After you create the window, use `GetNewControl` to create the scroll bar. Then use `MoveControl`, `SizeControl`, `SetControlMaximum` and `SetControlValue` to adjust the size, location and settings. Finally, use `ShowControl` to display the control bar.

Most applications allow the user to change the size of windows, add information to the document and remove information from the document. It is therefore necessary, in your window handling code, to calculate a changing maximum setting based on the document's current size and its window's current size. For new documents which have no content to scroll, assign an initial value of 0 as the maximum setting (which will, as previously stated, make the scroll bars inactive). Thereafter, your window-handling code should set and maintain the maximum setting.

By convention, a scroll bar is 16 pixels wide; accordingly, there should be a sixteen-pixel 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. (If you do not specify 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 one-pixel lines for the rectangle enclosing the scroll bar. As shown at Fig 2, the outside lines of the scroll bar should overlap the lines of the window frame.

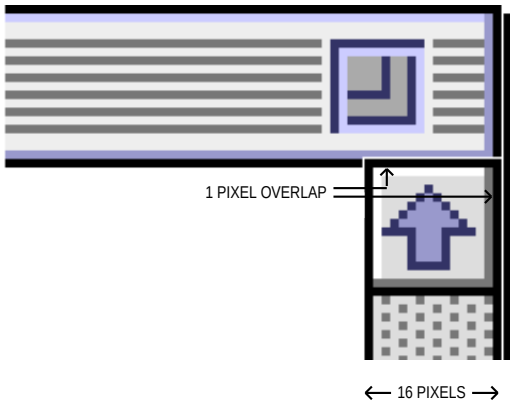

FIG 2 - CORRECT OVERLAP OF SCROLL BAR ON WINDOW FRAME

The following calculations[4] determine the rectangle for a vertical scroll bar:

Coordinate	Calculation
Top	Combined height of any items above the scroll bar - 1.
Left	Width of window - 15.
Bottom	Height of window - 14.
Right	Width of window +1.

The following calculations determine the rectangle for a horizontal scroll bar.

Coordinate	Calculation
Top	Height of window -15.
Left	Combined width of any items to the left of the scroll bar - 1.
Bottom	Height of window + 1.
Right	Width of window - 14.

[4]Do not include the title bar area in these calculations.

The top coordinate of a vertical scroll bar and the left coordinate of a horizontal scroll bar is -1 unless your application uses part of the window's typical scroll bar area for displaying information or specifying additional controls.

Just as the maximum settings change when the user resizes a document's window, so too do the scroll bar's coordinate locations change when the user resizes the window. The initial maximum settings and location, as specified in the `'CNTL'` resource, must therefore be changed dynamically by the application as required. Typically, this is achieved by storing handles to each scroll bar in a document record associated with the window and then using Control Manager routines to change control settings.

Creating Pop-Up Menus

The values you specify in a `'CNTL'` resource for a pop-up menu differ from those you supply in `'CNTL'` resources for other controls. An example of such a resource, in Rez input format, is as follows:

```
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. */
  popupMenuProc,       /* Control definition ID. */
  0,                   /* Reference value .*/
  "Speed:"             /* Control title. */
};
```

Rectangle. Fig 3 illustrates the rectangle for this pop-up menu.

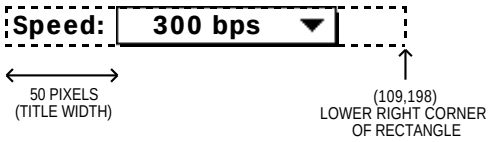

FIG 3 - DIMENSIONS OF SAMPLE POP-UP MENU

Title Position. The example `'CNTL'` resource specifies the title position in place of the initial control setting used for other types of controls. The example uses the `popupTitleLeftJust` constant to specify the position of the control title. Constants (and their values) which inform the Control Manager how to draw the menu's title are as follows:

Constant	Value
popupTitleBold	0x0100
popupTitleItalic	0x0200
popupTitleUnderline	0x0400
popupTitleOutline	0x0800
popupTitleShadow	0x1000
popupTitleCondense	0x2000
popupTitleExtend	0x4000
popupTitleNoStyle	0x0800
popupTitleLeftJust	0x0000
popupTitleCenterJust	0x0001
popupTitleRightJust	0x00FF

Title Width. The example `'CNTL'` resource specifies the width of the control title in place of the maximum setting used for other types of controls.

Menu Resource ID. The example `'CNTL'` resource specifies the appropriate `'MENU'` resource ID in place of the minimum setting used for other types of controls. The `'MENU'` resource provides the pop-up menu's items

Control Definition ID. You can specify a different control definition ID by adding any or all of the following constants to the `popupMenuProc` constant:

Constant	Setting	Description
popupFixedWidth	0x0001	Uses a constant control width, that is, does not resize the menu horizontally to fit long menu items. If a menu item does not fit in the space provided, it is truncated to fit and the ellipsis character (…) is appended at the end.
popupUseAddResMenu	0x0004	Gets menu items from a resource other than a 'MENU' resource. The control definition function will interpret the value in the contrlRfCon field of the control record as a value of type ResType. The control definition function uses AppendResMenu to add resources of that type to the menu.
popupUseWFont	0x0008	Uses the font of the specified window. The control definition function draws the pop-up menu title using the font and size of the window containing the control.

Reference Value. The Control Manager assigns the reference value to the control record's contrlRfCon field. When you create pop-up menus, your application should store handles for them, typically in a record pointed to by the contrlRfCon field of the window record. Storing these handles allows your application to respond to user's choices in pop-up menus.[5]

Menu Items and Control Values

When it creates the control, GetNewControl assigns the item number of the first menu item to the contrlValue field of the control record and sets the contrlMax field to the number of items in the pop-up menu. When the user chooses a different menu item, the Control Manager changes the contrlValue field to that item number.

Adding Resource Names as Items

If you specify popupUseAddResMenu as a variation code, the Control Manager coerces the value in the contrlRfCon field to the type ResType and then uses AppendResMenu to add items of that type. For example, if you specify a reference value of type (SInt32) 'FONT', the control definition function appends a list of fonts installed in the system to the menu associated with the pop-up menu.

Note that, after the control has been created, your application can use the contrlRfCon field for whatever purpose it requires.

Menu Width Adjustment

Whenever the pop-up menu is redrawn, its control definition function calls CalcMenuSize to calculate the size of the menu associated with the control (to allow for item additions and deletions). 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. Your application can override this behaviour by adding the popupFixedWidth variation code to the pop-up control definition ID.

Handling Mouse Events in Controls

Overview

For mouse events in controls, you usually perform the following tasks:

- Use FindWindow to determine the window in which the mouse-down event occurred.
- If the mouse-down event occurred in the content region of the active window, use FindControl to determine whether the event occurred in a control and, if so, which control.

[5] You should not use the Menu Manager function GetMenuHandle to obtain a handle to a menu associated with a pop-up menu control. If necessary, you can obtain a menu handle (and a menu ID) of a pop-up menu by dereferencing the contrlData field of the pop-up menu's control record. That field is a handle to a block of private information. For pop-up menus, it is a handle to a pop-up private data record.

- Call `TrackControl` to handle user interaction for the control as long as the user holds the mouse button down. The `actionProc` parameter passed to `TrackControl` should be as follows:
    - `NULL` for the scroll box and other standard controls.
    - For scroll arrows and gray areas of scroll bars, an application-defined **action procedure** which causes the document to scroll as long as the user holds the mouse button down.
    - `(ControlActionUPP) -1` for pop-up menus. This causes `TrackControl` to use the action procedure defined within the pop-up control definition function.
- When `TrackControl` reports that the user has released the mouse button with the cursor in a control, respond appropriately, that is:
    - Perform the task identified by the button title if the cursor is over a button.
    - Toggle the value of the checkbox when the cursor is over a checkbox. (The Control Manager then redraws 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 a 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 a 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.

Determining a Mouse-Down Event in a 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 that control's part. (When the mouse-down occurs in an invisible or inactive control, or when the cursor is not in a control, `FindControl` sets the control handle to `NULL` and returns 0 as its part code.)

A part code is an integer from 1 to 253. Part codes are assigned to a control by its control definition function. The standard control definition functions define the following part codes:

Constant	Old Name	Part Code	Control Part
`kControlButtonPart`	`inButton`	10	Button.
`kControlCheckBoxPart`	`inCheckBox`	11	Entire checkbox or radio button.
`kControlUpButtonPart`	`inUpButton`	20	Up scroll arrow (vertical scroll bar). Left scroll arrow (horizontal scroll bar).
`kControlDownButtonPart`	`inDownButton`	21	Down scroll arrow (vertical scroll bar). Right scroll arrow (horizontal scroll bar).
`kControlPageUpPart`	`InPageUp`	22	Gray area above scroll box (vertical scroll bar). Gray area to left of scroll box (horizontal scroll bar).
`kControlPageDownPart`	`inPageDown`	23	Gray area below scroll box (vertical scroll bar). Gray area to right of scroll box (horizontal scroll bar).
`kControlIndicatorPart`	`inThumb`	129	Scroll box.

The pop-up menu definition function does not define part codes for pop-up menus. Instead, and as previously stated, your application should store the handles for your pop-up menus when you create them and then test the handles you store against the handles returned by `FindControl`.

Tracking the Cursor in a Control

After calling `FindControl` to determine that the user pressed the mouse button while the cursor was in a control, call `TrackControl` to follow and respond to the user's movements and to determine the control part.

You can also use an action procedure to undertake additional actions as long as the user holds the mouse button down. Typically, action procedures are used to continuously scroll the window's contents while the cursor is on a scroll arrow. As previously stated, you pass a pointer to this action procedure as the third parameter in the `TrackControl` call.

The `TrackControl` function returns the control's part code if the user releases the mouse button while the cursor is still inside the control part, or 0 if the cursor is outside the control part when the button is released. Your application should then respond appropriately to a mouse-up event in that part.

Determining and Changing Control Settings

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

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 `MoveControl` and `SizeControl` to move and resize the scroll bars appropriately.

Your application can use `GetControlValue` to determine the current setting of a control, and it can use `GetControlMaximum` to determine a control's maximum setting. `SetControlValue` is used to change a control's setting and possibly redraw the scroll box accordingly. `SetControlMaximium` is used to change a control's maximum setting and to redraw the scroll box accordingly.

Moving and Resizing Scroll Bars

Your application must be able to size and move scroll bars dynamically in response to the user resizing your windows. The steps involved are:

- Resize the window.
- Use `HideControl` to make each scroll bar invisible.
- Use `MoveControl` to move the scroll bars to the appropriate edges of the window.
- Use `SizeControl` to lengthen or shorten each scroll bar as appropriate.
- Recalculate the maximum settings for the scroll bars and use `SetControlMaximum` to update the settings and to redraw the scroll boxes appropriately.
- Use `ShowControl` to make each scroll bar visible at its new location.

Each of the functions involved require a handle to the relevant scroll bar. When your application creates a window, it should store handles for each scroll bar in a document record associated with that window.

Scrolling Operations With Scroll Bars

Scrolling Basics

Spatial Relationships - Document, Window, and Scroll Bar

Spatial relationships between a document and a window, and their representation in a scroll bar, are shown at Fig 4.

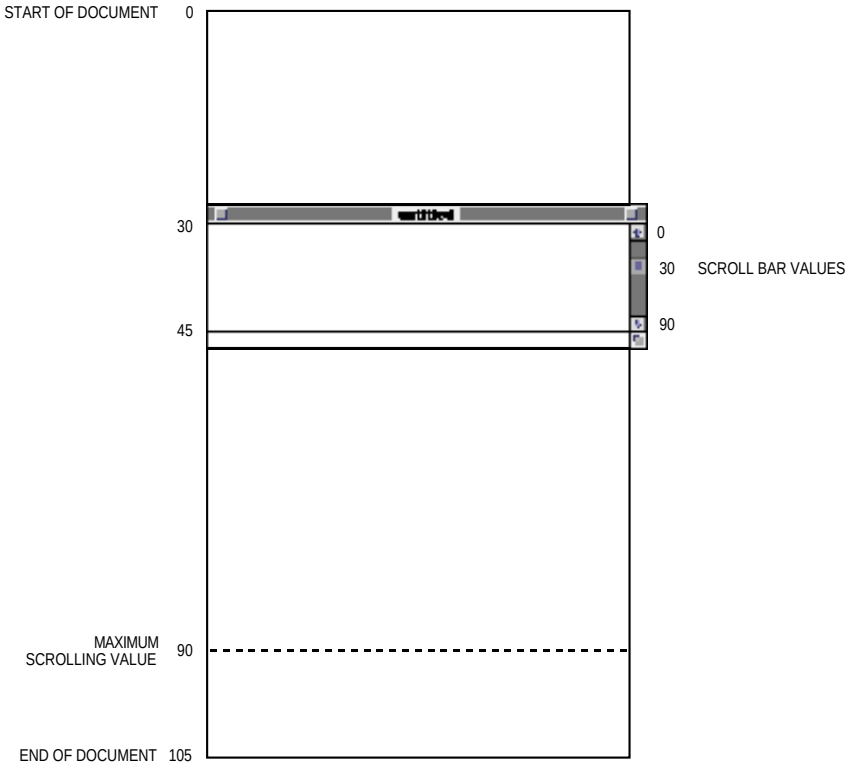

FIG 4 - SPATIAL RELATIONSHIP BETWEEN A DOCUMENT AND A WINDOW, AND THEIR REPRESENTATION IN A SCROLL BAR

Distance and Direction to Scroll

When the user scrolls a document using scroll bars, your application must first determine the distance and direction to scroll. The distance to scroll is as follows:

- When the user drags the scroll box to a new location, your application should scroll a corresponding distance in the document.
- When the user clicks on a scroll arrow, your application must determine an appropriate amount to scroll. Word processor applications typically scroll one line of text vertically, and horizontally by the average character width. Graphics applications typically scroll to display an entire object.
- When the user clicks in the gray area, your application must determine an appropriate amount to scroll. Typically, applications scroll by a distance of just less than the height or width of the window.[6]

[6]To determine this height and width, you can use the `contrlOwner` field of the scroll bar's control record, which contains a pointer to a window record.

The direction to scroll is determined by whether the scrolling distance is expressed as a positive or negative number. For example, when the user scrolls from the beginning of a document to a line 200 pixels down, the scrolling distance is -200 pixels on the vertical scroll bar.

Scrolling the Pixels

With the distance and direction to scroll determined, the next step is to scroll the pixels displayed in the window by that distance and in that direction. Typically, `ScrollRect` is used for that purpose.

Moving the Scroll Box

If the user did not effect the scroll using the scroll box, the scroll box must then be repositioned using `SetControlValue`.

Updating the Window

The final step is to either call a routine which generates an update event or directly call your application's update function. Your application's update function should call `UpdateControls` (to update the scroll bars) and redraw the appropriate part of the document in the window.

Scrolling Example

Half the complexity of scrolling lays in ensuring that that part of the document which is displayed in the window correlates with the scroll bar control value, and vice versa, at all times.

Consider the left-top of Fig 5, which illustrates the situation where the user has just opened an existing document. The document consists of 35 lines of monostyled text and the line height throughout is 10 pixels. The document is, therefore, 350 pixels long. When the user opens the document, the window origin is identical to the upper-left point of the document's space, that is, both are at (0,0).

In this example, the window displays 15 lines of text, which amounts to 150 pixels. Hence the maximum setting for the scroll bar is equivalent to 200 pixels down in the document. (As shown at Fig 2, a vertical scroll bar's maximum setting equates to the length of the document minus the height of the window.)

Now assume that the user drags the scroll box about halfway down the vertical scroll bar. Because the user wishes to scroll down, your application must move the text of the document up. Moving a document up in response to a user's request to scroll down requires a negative scrolling value.

Your application, using `GetControlValue`, determines that the scroll bar's control value is 100 and that it must therefore move the document up by 100 pixels. It then uses `ScrollRect` to shift the bits displayed in the window by a distance of -100 pixels (that is, 10 lines of text). As shown at the top-right of Fig 5, five lines from the bottom of the previous window display now appear at the top of the window. Your application adds the rest of the window to an update region for later updating.

Note that `ScrollRect` does not 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 the purposes of updating the window, you can think of this as changing the coordinates of the entire document, as is illustrated at the right-top of Fig 5.) In terms of the window's local coordinate system, then, the upper left corner of the document is now at (-100,0).

To facilitate updating of the window, `SetOrigin` must now be used to change the local coordinate system of the window so that the application can treat the upper left corner of the document as again lying at (0,0). This restoration of the document's original coordinate space makes it easier for the application to determine which lines of the document to draw in the update region of the window. (See bottom-left of Fig 5.)

Your application should now update the window by drawing lines 16 to 24, which it stores in its document record as beginning at (160,0) and ending at (250,0).

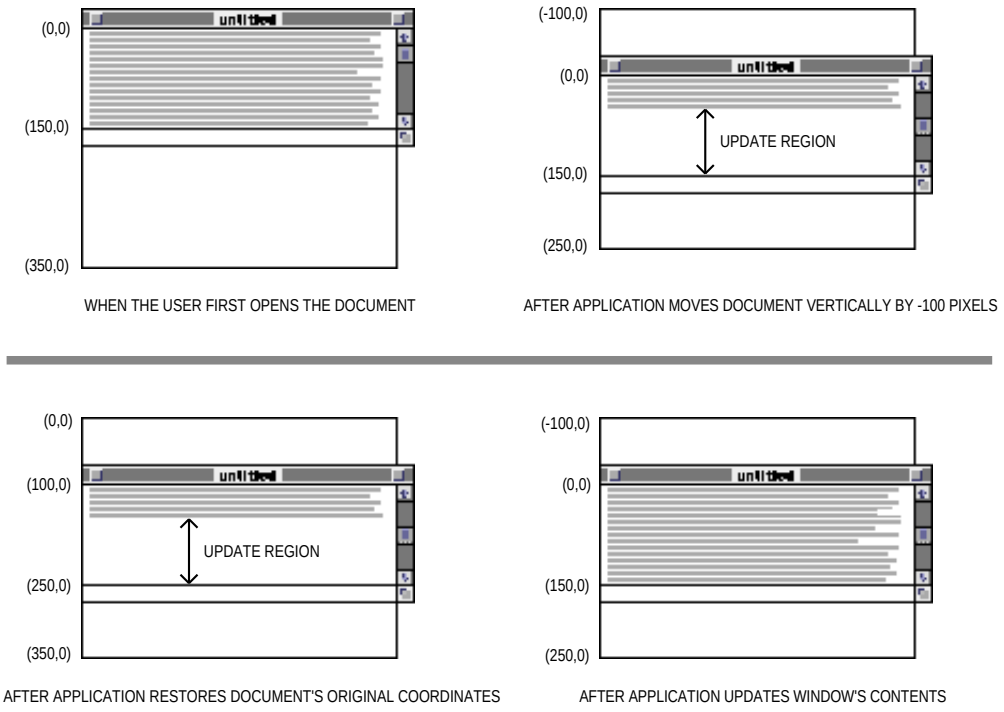

FIG 5 - SCROLLING A DOCUMENT IN A WINDOW

Finally, because the Window and Control Managers always assume that the window's upper-left point is at (0,0) when they draw in the window, the window origin cannot be left at (100,0). Accordingly, the application must use `SetOrigin` to reset it to (0,0) after performing its own drawing, (See bottom-right of Fig 5.)

To summarise:

- The user dragged the scroll box about half way down the vertical scroll bar. The application determined that this distance amounted to a scroll of -100 pixels.
- The application passed this distance to `ScrollRect`, which shifted the bits in the window 100 pixels upwards and created an update region in the vacated area of the window.
- The application passed the vertical scroll bar's current setting (100) in a parameter to `SetOrigin` so that the document's local coordinates were used when the update region of the window was redrawn. This changed the window's origin to (100,0).
- The application drew the text in the update region.
- The application reset the window's origin to (0,0)

Alternative to `SetOrigin`

There are alternatives to the `SetOrigin` methodology. `SetOrigin` simply helps you to offset the window's origin by the scroll bar's 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 at (0,0) and instead offset the items in your document, using `OffsetRect`, by an amount equal to the scroll bar's settings.

Scrolling a TextEdit Document

TextEdit is a collection of routines and data structures which you can use to provide your application with basic text editing capabilities. Chapter 17 — Text and TextEdit addresses, amongst other things, the scrolling of TextEdit documents.

Scrolling Using the List Manager

For scrolling lists of graphic or textual information, your application can use the List Manager to implement scroll bars. (See Chapter 18 — Lists and Custom List Definition Functions .)

Main Control Manager Constants, Data Types and Routines

Constants

Control Definition IDs

```
pushButProc             = 0
checkBoxProc            = 1
radioButProc            = 2
scrollBarProc           = 16
popupMenuProc           = 1008

useWFont                = 8         Add to pushButProc, checkBoxProc, radioButProc, to
                                    control title in the window font.
```

Pop-up Menu Variation Codes

```
popupFixedWidth         = 1 << 0
popupVariableWidth      = 1 << 1
popupUseAddResMenu      = 1 << 2
popupUseWFont           = 1 << 3    Add to popupMenuProc to display title in the window font.
```

Pop-up Title Characteristics

```
popupTitleBold          = 1 << 8
popupTitleItalic        = 1 << 9
popupTitleUnderline     = 1 << 10
popupTitleOutline       = 1 << 11
popupTitleShadow        = 1 << 12
popupTitleCondense      = 1 << 13
popupTitleExtend        = 1 << 14
popupTitleNoStyle       = 1 << 15
popupTitleLeftJust      = 0x00000000
popupTitleCenterJust    = 0x00000001
popupTitleRightJust     = 0x000000FF
```

Part Codes

```
                                    Old Names
kControlNoPart          = 0
kControlLabelPart       = 1         inLabel
kControlMenuPart        = 2         inMenu
kControlTrianglePart    = 4         inTriangle
kControlButtonPart      = 10        inButton
kControlCheckBoxPart    = 11        inCheckBox
kControlRadioButtonPart = 11
kControlUpButtonPart    = 20        inUpButton
kControlDownButtonPart  = 21        inDownButton
kControlPageUpPart      = 22        inPageUp
kControlPageDownPart    = 23        inPageDown
kControlIndicatorPart   = 129       inThumb
kControlDisabledPart    = 254
kControlInactivePart    = 255
```

Control Color Table Part Codes

```
cFrameColor             = 0
cBodyColor              = 1
```

```
cTextColor              = 2
cThumbColor             = 3
```

Data Types

```
typedef SInt16 ControlPartCode;
```

Control Record

```
struct ControlRecord
{
  ControlRef       nextControl;    // Next Control.
  WindowRef        contrlOwner;    // Control's window.
  Rect             contrlRect;     // Rectangle.
  UInt8            contrlVis;      // 255 if visible, else 0.
  UInt8            contrlHilite;   // Highlight state.
  SInt16           contrlValue;    // Current setting.
  SInt16           contrlMin;      // Minimum Setting.
  SInt16           contrlMax;      // Maximum setting.
  Handle           contrlDefProc;  // Control definition function.
  Handle           contrlData;     // Data used by contrlDefProc.
  ControlActionUPP contrlAction;   // Action procedure.
  SInt32           contrlRfCon;    // Reference constant.
  Str255           contrlTitle;    // Title.
};

typedef struct ControlRecord ControlRecord;
typedef ControlRecord *ControlPtr, **ControlHandle;

typedef ControlHandle ControlRef;
```

Auxiliary Control Record

```
struct AuxCtlRec
{
  Handle           acNext;         // Leads to next control list.
  ControlRef       acOwner;        // Handle to control that owns this auxiliary control.
  CCTabHandle      acCTable;       // Handle to inividial control's colour table.
  SInt16           acFlags;        // Reserved flag field.
  SInt32           acReserved;     // (Reserved.)
  SInt32           acRefCon;       // Application's reference constant.
};

typedef struct AuxCtlRec AuxCtlRec;
typedef AuxCtlRec *AuxCtlPtr, **AuxCtlHandle;
```

Control Color Table Record

```
struct CtlCTab
{
  SInt32           ccSeed;         // Unused, value = 0.
  SInt16           ccRider;        // Unused, value = 0.
  SInt16           ctSize;         // Elements in control's colour table minus 1.
  ColorSpec        ctTable[4];     // Address of ColorSpec records.
};

typedef struct CtlCTab CtlCTab;
typedef CtlCTab *CCTabPtr, **CCTabHandle;
```

Routines

Note: Some Control Manager routines can be accessed using more than one spelling of the routine's name, depending on the header files supported by your development environment. The following reflects the newest spellings, as specified in version 2.1 of the Universal Headers.

Creating Controls

```
ControlRef       NewControl(WindowRef theWindow,const Rect *boundsRect,ConstStr255Param
                 title,Boolean visible,SInt16 value,SInt16 min,SInt16 max,SInt16 procID,
                 SInt32 refCon);
ControlRef       GetNewControl(SInt16 controlID,WindowRef owner);
```

Drawing Controls

```
void            ShowControl(ControlRef theControl);
void            UpdateControls(WindowRef theWindow,RgnHandle updateRgn);
void            DrawControls(WindowRef theWindow);
void            Draw1Control(ControlRef theControl);
```

Handling Mouse Events in Controls

```
ControlPartCode  FindControl(Point thePoint,WindowRef theWindow,ControlRef *theControl);
ControlPartCode  TrackControl(ControlRef theControl,Point thePoint,ControlActionUPP
                 actionProc);
ControlPartCode  TestControl(ControlRef theControl, PointthePt);
```

Changing Control Settings and Display

```
void            SetControlValue(ControlRef theControl,SInt16 theValue);
void            SetControlMinimum(ControlRef theControl,SInt16 newMinimum);
void            SetControlMaximum(ControlRef theControl,SInt16 newMaximum);
void            SetControlTitle(ControlRef theControl,ConstStr255Param title);
void            HideControl(ControlRef theControl);
void            MoveControl(ControlRef theControl,SInt16 h,SInt16 v);
void            SizeControl(ControlRef theControl,SInt16 w,SInt16 h);
void            HiliteControl(ControlRef theControl,ControlPartCode hiliteState);
void            DragControl(ControlRef theControl,Point startPt,const Rect *limitRect,
                const Rect *slopRect,DragConstraint axis);
void            SetControlAction(ControlRef theControl,ControlActionUPP actionProc);
void            SetControlColor(ControlRef theControl,CCTabHandle newColorTable);
```

Determining Control Values

```
SInt16          GetControlValue(ControlRef theControl);
SInt16          GetControlMinimum(ControlRef theControl);
SInt16          GetControlMaximum(ControlRef theControl);
void            GetControlTitle(ControlRef theControl,Str255 title);
SInt32          GetControlReference(ControlRef theControl);
void            SetControlReference(ControlRef theControl,SInt32 data);
ControlActionUPP GetControlAction(ControlRef theControl);
SInt16          GetControlVariant(ControlRef theControl);
```

Removing Controls

```
void            DisposeControl(ControlRef theControl);
void            KillControls(WindowRef theWindow);
```

Demonstration Program 1

```
1   // ############################################################################################
2   // Controls1.c
3   // ############################################################################################
4   //
5   // This program opens a zoomDocProc window containing:
6   //
7   // •  A pop-up menu.
8   //
9   // •  Three radio buttons.
10  //
11  // •  Two checkboxes.
12  //
13  // •  One button.
14  //
15  // •  Vertical and horizontal scroll bars.
16  //
17  // The pop-up menu, radio buttons, checkboxes, and button work correctly except that the
18  // control values are not used for any specific purpose.
19  //
20  // The scroll bars are moved and resized when the user resizes or zooms the window;
21  // however, no action is taken when the scroll box is moved or the scroll arrows or gray
22  // areas are clicked.
23  //
24  // The program utilises the following resources:
25  //
```

```
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit menus and a pop-up
//    menu (preload, non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially not visible).
//
// •  'CNTL' resources for the pop-up menu, radio buttons, checkboxes, button and
//    scroll bars (preload, purgeable) (initially visible).
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch,
//    and is32BitCompatible flags set.
//
// ############################################################################################

// ……… includes

#include <Fonts.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <SegLoad.h>
#include <ToolUtils.h>
#include <Devices.h>

// ……… defines

#define rMenubar    128
#define rNewWindow  128
#define mApple      128
#define  iAbout     1
#define mFile       129
#define  iQuit      11
#define mEdit       130
#define cTimeZone   128
#define  pSydney    1
#define  pNewYork   2
#define  pLondon    3
#define  pRome      4
#define cRed        129
#define cWhite      130
#define cBlue       131
#define cShowgrid   132
#define cShowrulers 133
#define cButton     134
#define cVScrollbar 135
#define cHScrollbar 136
#define MAXLONG     0x7FFFFFFF

// ……… typedefs

typedef struct
{
  ControlHandle popupControlHdl;
  ControlHandle redHdl;
  ControlHandle whiteHdl;
  ControlHandle blueHdl;
  ControlHandle showGridHdl;
  ControlHandle showRulersHdl;
  ControlHandle okButtonHdl;
  ControlHandle vScrollbarHdl;
  ControlHandle hScrollbarHdl;
} DocRec;

typedef DocRec **DocRecHandle;

// ……… global variables

Boolean gDone;
Boolean gInBackground;

// ……… function prototypes

void  main                  (void);
void  doInitManagers        (void);
void  doGetControls         (WindowPtr);
void  doEvents              (EventRecord *);
void  doMouseDown           (EventRecord *);
void  doMenuChoice          (SInt32);
```

```
103  void  doInContent             (EventRecord *,WindowPtr);
104  void  doPopupMenuChoice       (SInt16);
105  void  doControls              (ControlHandle,DocRecHandle);
106  void  doUpdate                (EventRecord *);
107  void  doActivate              (EventRecord *);
108  void  doActivateWindow        (WindowPtr,Boolean);
109  void  doOSEvent               (EventRecord *);
110  void  doAdjustScrollBars      (WindowPtr);
111  void  doEraseGrowIcon         (WindowPtr);
112
113  // ############################################################################## main
114
115  void  main(void)
116  {
117    Handle        menubarHdl;
118    MenuHandle    menuHdl;
119    WindowPtr     windowPtr;
120    DocRecHandle  docRecHdl;
121    EventRecord   eventRec;
122
123    // ……… initialize managers
124
125    doInitManagers();
126
127    // ……… set up menu bar and menus
128
129    menubarHdl = GetNewMBar(rMenubar);
130    if(menubarHdl == NULL)
131      ExitToShell();
132    SetMenuBar(menubarHdl);
133    DrawMenuBar();
134
135    menuHdl = GetMenuHandle(mApple);
136    if(menuHdl == NULL)
137      ExitToShell();
138    else
139      AppendResMenu(menuHdl,'DRVR');
140
141    // ……… open a window
142
143    if(!(windowPtr = GetNewWindow(rNewWindow,NULL,(WindowPtr)-1)))
144      ExitToShell();
145    SetPort(windowPtr);
146
147    // …… get block for document record, assign handle to window record refCon field
148
149    if(!(docRecHdl = (DocRecHandle) NewHandle(sizeof(DocRec))))
150      ExitToShell();
151
152    SetWRefCon(windowPtr,(SInt32) docRecHdl);
153
154    // ……… get controls, adjust scroll bars and show window
155
156    doGetControls(windowPtr);
157    doAdjustScrollBars(windowPtr);
158
159    ShowWindow(windowPtr);
160
161    // ……… enter eventLoop
162
163    gDone = false;
164
165    while(!gDone)
166    {
167      if(WaitNextEvent(everyEvent,&eventRec,MAXLONG,NULL))
168        doEvents(&eventRec);
169    }
170  }
171
172  // ########################################################################## doInitManagers
173
174  void  doInitManagers(void)
175  {
176    MaxApplZone();
177    MoreMasters();
178
179    InitGraf(&qd.thePort);
```

```
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(NULL);

  InitCursor();
  FlushEvents(everyEvent,0);
}

// ############################################################################ doGetControls

void  doGetControls(WindowPtr windowPtr)
{
  DocRecHandle docRecHdl;

  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));

  if(!((*docRecHdl)->popupControlHdl = GetNewControl(cTimeZone,windowPtr)))
    ExitToShell();

  if(!((*docRecHdl)->redHdl = GetNewControl(cRed,windowPtr)))
    ExitToShell();
  if(!((*docRecHdl)->whiteHdl = GetNewControl(cWhite,windowPtr)))
    ExitToShell();
  if(!((*docRecHdl)->blueHdl = GetNewControl(cBlue,windowPtr)))
    ExitToShell();

  if(!((*docRecHdl)->showGridHdl = GetNewControl(cShowgrid,windowPtr)))
    ExitToShell();
  if(!((*docRecHdl)->showRulersHdl = GetNewControl(cShowrulers,windowPtr)))
    ExitToShell();

  if(!((*docRecHdl)->okButtonHdl = GetNewControl(cButton,windowPtr)))
    ExitToShell();

  if(!((*docRecHdl)->vScrollbarHdl = GetNewControl(cVScrollbar,windowPtr)))
    ExitToShell();
  if(!((*docRecHdl)->hScrollbarHdl = GetNewControl(cHScrollbar,windowPtr)))
    ExitToShell();
}

// ################################################################################ doEvents

void  doEvents(EventRecord *eventRecPtr)
{
  switch(eventRecPtr->what)
  {
    case mouseDown:
      doMouseDown(eventRecPtr);
      break;

    case updateEvt:
      doUpdate(eventRecPtr);
      break;

    case activateEvt:
      doActivate(eventRecPtr);
      break;

    case osEvt:
      doOSEvent(eventRecPtr);
      HiliteMenu(0);
      break;
  }
}

// ############################################################################ doMouseDown

void  doMouseDown(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;
  SInt16    partCode;
  Rect      growRect;
  SInt32    newSize;

  partCode = FindWindow(eventRecPtr->where,&windowPtr);
```

```
257
258     switch(partCode)
259     {
260       case inMenuBar:
261         doMenuChoice(MenuSelect(eventRecPtr->where));
262         break;
263
264       case inSysWindow:
265         SystemClick(eventRecPtr,windowPtr);
266         break;
267
268       case inContent:
269         if(windowPtr != FrontWindow())
270           SelectWindow(windowPtr);
271         else
272           doInContent(eventRecPtr,windowPtr);
273         break;
274
275       case inDrag:
276         DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
277         break;
278
279       case inGoAway:
280         if(TrackGoAway(windowPtr,eventRecPtr->where) == true)
281           gDone = true;
282         break;
283
284       case inGrow:
285         growRect = qd.screenBits.bounds;
286         growRect.top  = 200;
287         growRect.left = 275;
288         newSize = GrowWindow(windowPtr,eventRecPtr->where,&growRect);
289         if(newSize != 0)
290         {
291           doEraseGrowIcon(windowPtr);
292           SizeWindow(windowPtr,LoWord(newSize),HiWord(newSize),true);
293           doAdjustScrollBars(windowPtr);
294         }
295         break;
296
297       case inZoomIn:
298       case inZoomOut:
299         if(TrackBox(windowPtr,eventRecPtr->where,partCode))
300         {
301           SetPort(windowPtr);
302           EraseRect(&windowPtr->portRect);
303           ZoomWindow(windowPtr,partCode,false);
304           InvalRect(&windowPtr->portRect);
305           doAdjustScrollBars(windowPtr);
306         }
307         break;
308     }
309   }
310
311   // ############################################################################ doMenuChoice
312
313   void  doMenuChoice(SInt32 menuChoice)
314   {
315     SInt16  menuID, menuItem;
316     Str255  itemName;
317     SInt16  daDriverRefNum;
318
319     menuID = HiWord(menuChoice);
320     menuItem = LoWord(menuChoice);
321
322     if(menuID == 0)
323       return;
324
325     switch(menuID)
326     {
327       case mApple:
328         if(menuItem == iAbout)
329           SysBeep(10);
330         else
331         {
332           GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
333           daDriverRefNum = OpenDeskAcc(itemName);
```

```
      }
      break;

    case mFile:
      if(menuItem == iQuit)
        gDone = true;
      break;
  }

  HiliteMenu(0);
}

// ########################################################################## doInContent

void  doInContent(EventRecord *eventRecPtr,WindowPtr windowPtr)
{
  ControlHandle controlHdl;
  SInt16        controlValue;
  DocRecHandle  docRecHdl;

  GlobalToLocal(&eventRecPtr->where);

  if(FindControl(eventRecPtr->where,windowPtr,&controlHdl))
  {
    docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
    if(controlHdl == (*docRecHdl)->popupControlHdl)
    {
      TrackControl(controlHdl,eventRecPtr->where,(ControlActionUPP) -1);
      controlValue = GetControlValue(controlHdl);
      doPopupMenuChoice(controlValue);
    }
    else
    {
      if(TrackControl(controlHdl,eventRecPtr->where,NULL))
        doControls(controlHdl,docRecHdl);
    }
  }
}

// ###################################################################### doPopupMenuChoice

void  doPopupMenuChoice(SInt16 controlValue)
{
  switch(controlValue)
  {
    case pSydney:
      // Action as appropriate.
      break;

    case pNewYork:
      // Action as appropriate.
      break;

    case pLondon:
      // Action as appropriate.
      break;

    case pRome:
      // Action as appropriate.
      break;
  }

  SysBeep(10);
}

// ########################################################################## doControls

void  doControls(ControlHandle controlHdl,DocRecHandle docRecHdl)
{
  if(controlHdl == (*docRecHdl)->redHdl || controlHdl == (*docRecHdl)->whiteHdl ||
                   controlHdl == (*docRecHdl)->blueHdl)
  {
    SetControlValue((*docRecHdl)->redHdl,0);
    SetControlValue((*docRecHdl)->whiteHdl,0);
    SetControlValue((*docRecHdl)->blueHdl,0);
    SetControlValue(controlHdl,1);
  }
```

```
411     else if(controlHdl == (*docRecHdl)->showGridHdl ||
412                           controlHdl == (*docRecHdl)->showRulersHdl)
413     {
414       SetControlValue(controlHdl,!GetControlValue(controlHdl));
415     }
416     else if(controlHdl == (*docRecHdl)->vScrollbarHdl ||
417                           controlHdl == (*docRecHdl)->hScrollbarHdl)
418     {
419       // Do scroll bars handling.
420     }
421     else
422     {
423       // Must be button.  Do button handling.
424     }
425
426     SysBeep(10);
427   }
428
429   // ############################################################################## doUpdate
430
431   void  doUpdate(EventRecord *eventRecPtr)
432   {
433     WindowPtr windowPtr;
434
435     windowPtr = (WindowPtr) eventRecPtr->message;
436
437     BeginUpdate(windowPtr);
438
439     if(!EmptyRgn(windowPtr->visRgn))
440     {
441       SetPort(windowPtr);
442       UpdateControls(windowPtr,windowPtr->visRgn);
443       DrawGrowIcon(windowPtr);
444     }
445
446     EndUpdate(windowPtr);
447   }
448
449   // ############################################################################ doActivate
450
451   void  doActivate(EventRecord *eventRecPtr)
452   {
453     WindowPtr windowPtr;
454     Boolean   becomingActive;
455
456     windowPtr = (WindowPtr) eventRecPtr->message;
457
458     becomingActive = ((eventRecPtr->modifiers & activeFlag) == activeFlag);
459
460     doActivateWindow(windowPtr,becomingActive);
461   }
462
463   // ######################################################################## doActivateWindow
464
465   void  doActivateWindow(WindowPtr windowPtr,Boolean becomingActive)
466   {
467     DocRecHandle  docRecHdl;
468     SInt16        hiliteState;
469
470     docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
471
472     if(becomingActive)
473       hiliteState = 0;
474     else
475       hiliteState = 255;
476
477     HiliteControl((*docRecHdl)->popupControlHdl,hiliteState);
478     HiliteControl((*docRecHdl)->redHdl,         hiliteState);
479     HiliteControl((*docRecHdl)->whiteHdl,       hiliteState);
480     HiliteControl((*docRecHdl)->blueHdl,        hiliteState);
481     HiliteControl((*docRecHdl)->showGridHdl,    hiliteState);
482     HiliteControl((*docRecHdl)->showRulersHdl,  hiliteState);
483     HiliteControl((*docRecHdl)->okButtonHdl,    hiliteState);
484     HiliteControl((*docRecHdl)->vScrollbarHdl,  hiliteState);
485     HiliteControl((*docRecHdl)->hScrollbarHdl,  hiliteState);
486   }
487
```

```
488 // ############################################################################ doOSEvent
489
490 void  doOSEvent(EventRecord *eventRecPtr)
491 {
492   switch((eventRecPtr->message >> 24) & 0x000000FF)
493   {
494     case suspendResumeMessage:
495       DrawGrowIcon(FrontWindow());
496       gInBackground = (eventRecPtr->message & resumeFlag) == 0;
497       doActivateWindow(FrontWindow(),!gInBackground);
498       break;
499
500     case mouseMovedMessage:
501       break;
502   }
503 }
504
505 // ######################################################################## doAdjustScrollBars
506
507 void  doAdjustScrollBars(WindowPtr windowPtr)
508 {
509   Rect          winRect;
510   DocRecHandle  docRecHdl;
511
512   docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
513
514   winRect = windowPtr->portRect;
515
516   HideControl((*docRecHdl)->vScrollbarHdl);
517   HideControl((*docRecHdl)->hScrollbarHdl);
518
519   MoveControl((*docRecHdl)->vScrollbarHdl,winRect.right - 15,winRect.top - 1);
520   MoveControl((*docRecHdl)->hScrollbarHdl,winRect.left -1,winRect.bottom -15);
521
522   SizeControl((*docRecHdl)->vScrollbarHdl,16, winRect.bottom - 13);
523   SizeControl((*docRecHdl)->hScrollbarHdl, winRect.right - 13,16);
524
525   ShowControl((*docRecHdl)->vScrollbarHdl);
526   ShowControl((*docRecHdl)->hScrollbarHdl);
527
528   DrawGrowIcon(windowPtr);
529 }
530
531 // ######################################################################## doEraseGrowIcon
532
533 void  doEraseGrowIcon(WindowPtr windowPtr)
534 {
535   Rect  growBoxRect;
536
537   SetPort(windowPtr);
538
539   growBoxRect = windowPtr->portRect;
540   growBoxRect.left = growBoxRect.right - 15;
541   growBoxRect.top = growBoxRect.bottom - 15;
542   EraseRect(&growBoxRect);
543 }
544
545 // ##############################################################################################
```

Demonstration Program 1 Comments

When this program is run, the user should:

- Click on the various controls, noting particularly that the radio button settings are mutually exclusive and that checkbox settings are not.

- Resize and zoom the window, noting that the scroll bars are moved and resized in response to those actions.

- Send the program to the background and bring it to the foreground, noting the changes to the appearance of the controls. (As a point of interest, users with a colour or grayscale monitor and a Macintosh on which Color QuickDraw is present will note that, when the controls are unhighlighted, the pop-up menu appears in a grey colour whereas the titles of the other controls appears in the gray pattern. If the window is opened

using GetNewCWindow, rather than GetNewWindow, the titles of these latter controls will appear in a grey colour. The control definition functions determine this behaviour.)

#define

Line 51-70 establish constants representing menu, window and control resource IDs, menu IDs and menu items. Line 71 defines MAXLONG as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

#typedef

At Lines 75-78, a data type for a document record is created. The document record structure comprises fields in which the handles to the control records for the various controls will be stored.

Global Variables

gDone is used to control termination of the program, which will occur when the user selects Quit from the File menu or clicks in the window's close box. gInBackground relates to foreground/background switching.

main

Within the main function, the system software managers are initialised (Line 125), the menu bar and drop-down menus are set up (Lines 129-139), and a zoomDocProc window is opened (Line 143).

At Line 149, a relocatable block the size of one document record is created. At Line 152, the handle to the block is assigned to the window record's refCon field.

At Line 156, a call is made to the application-defined function which creates the controls. Line 157 calls the application-defined function which resizes and locates the scroll bars according to the dimensions of the window's port rectangle. With the controls created, Line 159 makes the window visible.

The main event loop is then entered (Lines 163-169).

Note that error handling here and in other areas of this demonstration program is somewhat rudimentary. In the unlikely event that certain calls fail, ExitToShell is called to terminate the program.

doGetControls

The function doGetControls creates the controls from the various 'CNTL' resources. Firstly, at Line 196, the handle to the structure in which the handles to the control records will be stored is retrieved. Then, at Lines 198-219, calls to GetNewControl create a control record for each control, insert the record into the control list for the specified window and draw the control. At the same time, the handle to each control is assigned to the appropriate field of the window's document record.

doEvents

doEvents switches according to the event type reported.

doMouseDown

doMouseDown switches according to the window part in which a mouseDown event occurs.

If the window in which the mouse-down occurred is the front window (Line 269), and since all of the controls are located in the window's content region, a call to the application-defined function doInContent is made at Line 272.

Lines 284-295 handle re-sizing of the window, which is of particular significance to the scroll bars. GrowWindow (Line 288) follows the mouse cursor while the mouse button remains down, returning the new height and width of the window, or zero if no change was made. If a change was made (Line 289), an application-defined function is called to erase the grow box, SizeWindow is called to draw the window in its new size (Line 292), and an application-defined function is called at Line 293 to erase, move, resize and redraw the scroll bars.

Lines 297-307 handle window zooming, which is also of significance to the scroll bars. If the call to TrackBox at Line 299 returns a non-zero value, the window's content region is erased (Lines 301-302), ZoomWindow is called at Line 303 to redraw the window in its new state, InvalRect is called at Line 304 to add the entire content region to the update region, and an

application-defined function is called at Line 305 to erase, move, resize and redraw the scroll bars.

doMenuChoice

doMenuChoice handles user choices from the drop-down menus.

doInContent

doInContent further processes mouse-down events in the content region.

Line 354 converts the mouse coordinates in the event record's where field to the local coordinates required in the call to FindControl at Line 356.

If there is a control at the cursor location at which the mouse button is released, the control handle returned by the FindControl call at Line 356 is compared with the handle to the pop-up control stored in the window's document record (Line 359). If they match, TrackControl is called (Line 361) with the procPtr field set (ControlActionUPP) -1 so as to cause an action procedure within the control's control definition function to be repeatedly invoked while the mouse button remains down. When TrackControl returns, the control value is obtained by a call to GetControlValue (Line 362). For a pop-up menu, this value represents the menu item number. At Line 363, the menu item number is passed to an application-defined function which handles the menu choice.

If the control handle returned by the FindControl call does not match the pop-up control's handle (Line 365), the handle must be to one of the other controls. In this case, TrackControl is called (Line 367), with the procPtr field set to that required for a radio button, checkbox, button or scroll bar. If the cursor is still within the control when the mouse button is released, the handle to the control record found by FindControl, together with the handle to the window's document record, is passed to the application-defined function doControls (Line 368).

doPopupMenuChoice

doPopupMenuChoice switches according to the menu item number passed to it from Line 363.

doControls

doControls receives control and document record handles and switches according to whether the control handle matches one of the radio button handles, one of the checkbox handles or the button handle.

If the control handle matches one of the radio button handles (Line 403), the control values of all radio buttons are set to 0 before that for the selected control is set to 1 (Lines 406-409). If the control handle matches one of the checkboxes (Line 411), the control value for that control is flipped (Line 414). If the control handle matches that of one the scroll bars (Line 416), appropriate scrolling handling is invoked (Line 419).

If a match has still not been found, the handle must be a match for the button's handle (Line 421, in which case the appropriate action is taken (Line 423).

doUpdate

doUpdate is called whenever the application receives an update event for its window. Between the usual calls to BeginUpdate and EndUpdate, and if the window's visible region (which at that point equates to the update region as it was prior to the BeginUpdate call) is not empty, the window's graphics port is set as the current port for drawing (Line 441), UpdateControls is called at Line 442 to draw those controls intersecting the current visible region, and DrawGrowIcon is called to draw the grow icon (Line 443).

doActivate

doActivate is called whenever the application receives an activate event for its window. At Line 458, a variable is set to indicate whether the window is becoming active or is about to be made inactive. This variable is then passed in the call to an application-defined function doActivateWindow at Line 460.

doActivateWindow

doActivateWindow switches according to whether the specified window is becoming active or is about to be made inactive. (Actually, doActivateWindow will never be called by doActivate in this program because the program only opens one window. It will however, be called by the application-defined function doOSEvent.)

At Line 470, a handle to the window's document record is retrieved from the window record's refCon field. If the window is becoming active (Line 472), the variable hiliteState is assigned the value for un-dimming the controls and making them active. If the window is about to become inactive (Line 474), this same variable is assigned the value for dimming the controls and making them inactive. HiliteControl is then called for all controls (Lines 477-485).

doOSEvent

doOSEvent handles operating system events.

If the event is a suspend or resume event (Line 494), DrawGrowIcon is called at Line 495 to draw the grow icon in the appropriate state. A variable is then set to indicate whether the program is coming to the foreground or is about to be sent to the background (Line 496). This variable is passed in the call to doActivateWindow at Line 497. (Recall that the doesActivateOnFGSwitch flag is set in the 'SIZE' resource.)

doAdjustScrollBars

doAdjustScrollBars is called if the user resizes or zooms the window.

At Line 512, a handle to the window's document record is retrieved from the window record's refCon field. At Line 514, the coordinates representing the window's current content region are assigned to a Rect variable which will be used in calls to MoveControl and SizeControl.

Amongst other things, MoveControl and SizeControl both redraw the specified scroll bar. Since SizeControl will be called immediately after MoveControl, this will causes a very slight flickering of the scroll bars. To prevent this, the scroll bars will be hidden while these two functions are executing.

Lines 516-517 hide the scroll bars. The calls to MoveControl at Lines 519-520 erase the scroll bars, offset the contrlRect fields of their control records, and redraw the scroll bars within the offset rectangle. The calls to SizeControl at Lines 522-523 hide the scroll bars (in this program they are already hidden), adjust the contrlRect fields of their control records, and redraw the scroll bars within the new rectangle. Lines 525-526 show the scroll bars. Line 528 draws the grow icon.

doEraseGrowIcon

doEraseGrowIcon is called whenever the user resizes the window. It erases the size box.

Demonstration Program 2

```
1  // ##############################################################################################
2  // Controls2.c
3  // ##############################################################################################
4  //
5  // This program:
6  //
7  // •  Opens a noGrowDocProc window with a horizontal scrollbar.
8  //
9  // •  Allows the user to horizontally scroll a picture within the window using the
10 //    scroll box, the scroll arrows and the gray area.
11 //
12 // The program utilises the following resources:
13 //
14 // •  An 'MBAR' resource, and 'MENU' resources for Apple, File and Edit (preload, non-
15 //    purgeable).
16 //
17 // •  A 'WIND' resource (purgeable) (initially visible).
18 //
19 // •  An 'CNTL' resource for the horizontal scroll bar (purgeable).
20 //
21 // •  A 'PICT' resource containing the picture to be scrolled (non-purgeable).
22 //
23 // •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch,
24 //    and is32BitCompatible flags set.
25 //
26 // ##############################################################################################
27
28 // …… includes
```

```
29
30  #include <Fonts.h>
31  #include <Menus.h>
32  #include <TextEdit.h>
33  #include <Dialogs.h>
34  #include <SegLoad.h>
35  #include <ToolUtils.h>
36  #include <Devices.h>
37
38  // ..... defines
39
40  #define rMenubar    128
41  #define rNewWindow  128
42  #define rPicture    128
43  #define mApple      128
44  #define  iAbout     1
45  #define mFile       129
46  #define  iQuit      11
47  #define mEdit       130
48  #define cHScrollbar 128
49  #define MAXLONG     0x7FFFFFFF
50
51  // ..... typedefs
52
53  typedef struct
54  {
55    ControlHandle hScrollbarHdl;
56  } DocRec;
57
58  typedef DocRec **DocRecHandle;
59
60  // ..... global variables
61
62  Boolean   gDone;
63  Boolean   gInBackground;
64  Rect      gPictRect;
65  PicHandle gPictureHdl;
66
67  // ..... function prototypes
68
69  void          main               (void);
70  void          doInitManagers     (void);
71  void          doGetControl       (WindowPtr);
72  void          doGetPicture       (void);
73  void          doEvents           (EventRecord *);
74  void          doMouseDown        (EventRecord *);
75  void          doUpdate           (EventRecord *);
76  void          doActivate         (EventRecord *);
77  void          doActivateWindow   (WindowPtr,Boolean);
78  void          doOSEvent          (EventRecord *);
79  void          doMenuChoice       (SInt32);
80  void          doInContent        (EventRecord *,WindowPtr);
81  void          doScrollBars       (ControlPartCode,WindowPtr,ControlHandle,Point);
82  pascal void   actionProcedure    (ControlHandle,ControlPartCode);
83  void          doMoveScrollBox    (ControlHandle,SInt16);
84
85  // ############################################################################## main
86
87  void  main(void)
88  {
89    Handle        menubarHdl;
90    MenuHandle    menuHdl;
91    WindowPtr     windowPtr;
92    DocRecHandle  docRecHdl;
93    EventRecord   eventRec;
94
95    // ..... initialise managers
96
97    doInitManagers();
98
99    // ..... set up menu bar and menus
100
101   menubarHdl = GetNewMBar(rMenubar);
102   if(menubarHdl == NULL)
103     ExitToShell();
104   SetMenuBar(menubarHdl);
105   DrawMenuBar();
```

```
106
107     menuHdl = GetMenuHandle(mApple);
108     if(menuHdl == NULL)
109       ExitToShell();
110     else
111       AppendResMenu(menuHdl,'DRVR');
112
113     // ……… open a window
114
115     if(!(windowPtr = GetNewWindow(rNewWindow,NULL,(WindowPtr)-1)))
116       ExitToShell();
117
118     SetPort(windowPtr);
119
120     // ……… get block for document record, assign handle to window record refCon field
121
122     docRecHdl = (DocRecHandle) NewHandle(sizeof(DocRec));
123     SetWRefCon(windowPtr,(SInt32) docRecHdl);
124
125     // ……… get controls
126
127     doGetControl(windowPtr);
128
129     // ……… get picture
130
131     doGetPicture();
132
133     // ……… enter eventLoop
134
135     gDone = false;
136
137     while(!gDone)
138     {
139       if(WaitNextEvent(everyEvent,&eventRec,MAXLONG,NULL))
140         doEvents(&eventRec);
141     }
142   }
143
144   // ############################################################################ doInitManagers
145
146   void  doInitManagers(void)
147   {
148     MaxApplZone();
149     MoreMasters();
150
151     InitGraf(&qd.thePort);
152     InitFonts();
153     InitWindows();
154     InitMenus();
155     TEInit();
156     InitDialogs(NULL);
157
158     InitCursor();
159     FlushEvents(everyEvent,0);
160   }
161
162   // ############################################################################ doGetControl
163
164   void  doGetControl(WindowPtr windowPtr)
165   {
166     DocRecHandle docRecHdl;
167
168     docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
169
170     (*docRecHdl)->hScrollbarHdl = GetNewControl(cHScrollbar,windowPtr);
171   }
172
173   // ############################################################################ doGetPicture
174
175   void  doGetPicture(void)
176   {
177     gPictureHdl = GetPicture(rPicture);
178
179     gPictRect = (*gPictureHdl)->picFrame;
180
181     gPictRect.right -=  gPictRect.left;
182     gPictRect.left = 0;
```

```
  gPictRect.bottom -= gPictRect.top;
  gPictRect.top = 0;
}

// ############################################################################## doEvents

void  doEvents(EventRecord *eventRecPtr)
{
  switch(eventRecPtr->what)
  {
    case mouseDown:
      doMouseDown(eventRecPtr);
      break;

    case updateEvt:
      doUpdate(eventRecPtr);
      break;

    case activateEvt:
      doActivate(eventRecPtr);
      break;

    case osEvt:
      doOSEvent(eventRecPtr);
      HiliteMenu(0);
      break;
  }
}

// ############################################################################ doMouseDown

void  doMouseDown(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;
  SInt16    partCode;

  partCode = FindWindow(eventRecPtr->where,&windowPtr);

  switch(partCode)
  {
    case inMenuBar:
      doMenuChoice(MenuSelect(eventRecPtr->where));
      break;

    case inSysWindow:
      SystemClick(eventRecPtr,windowPtr);
      break;

    case inContent:
      if(windowPtr != FrontWindow())
        SelectWindow(windowPtr);
      else
        doInContent(eventRecPtr,windowPtr);
      break;

    case inDrag:
      DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
      break;

    case inGoAway:
      if(TrackGoAway(windowPtr,eventRecPtr->where) == true)
        gDone = true;
      break;
  }
}

// ############################################################################## doUpdate

void  doUpdate(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;
  DocRecHandle  docRecHdl;

  windowPtr = (WindowPtr)eventRecPtr->message;
  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));

  BeginUpdate(windowPtr);
```

```
260
261     if(!EmptyRgn(windowPtr->visRgn))
262     {
263       SetPort(windowPtr);
264       UpdateControls(windowPtr,windowPtr->visRgn);
265
266       SetOrigin(GetControlValue((*docRecHdl)->hScrollbarHdl),0);
267       DrawPicture(gPictureHdl,&gPictRect);
268       SetOrigin(0,0);
269     }
270
271     EndUpdate(windowPtr);
272   }
273
274   // ############################################################################## doActivate
275
276   void  doActivate(EventRecord *eventRecPtr)
277   {
278     WindowPtr windowPtr;
279     Boolean   becomingActive;
280
281     windowPtr = (WindowPtr) eventRecPtr->message;
282
283     becomingActive = ((eventRecPtr->modifiers & activeFlag) == activeFlag);
284
285     doActivateWindow(windowPtr,becomingActive);
286   }
287
288   // ########################################################################## doActivateWindow
289
290   void  doActivateWindow(WindowPtr windowPtr,Boolean becomingActive)
291   {
292     DocRecHandle  docRecHdl;
293
294     docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));
295
296     if(becomingActive)
297       HiliteControl((*docRecHdl)->hScrollbarHdl,0);
298     else
299       HiliteControl((*docRecHdl)->hScrollbarHdl,255);
300   }
301
302   // ################################################################################ doOSEvent
303
304   void  doOSEvent(EventRecord *eventRecPtr)
305   {
306     switch((eventRecPtr->message >> 24) & 0x000000FF)
307     {
308       case suspendResumeMessage:
309         gInBackground = (eventRecPtr->message & resumeFlag) == 0;
310         doActivateWindow(FrontWindow(),!gInBackground);
311         break;
312
313       case mouseMovedMessage:
314         break;
315     }
316   }
317
318   // ############################################################################## doMenuChoice
319
320   void  doMenuChoice(SInt32 menuChoice)
321   {
322     SInt16  menuID, menuItem;
323     Str255  itemName;
324     SInt16  daDriverRefNum;
325
326     menuID = HiWord(menuChoice);
327     menuItem = LoWord(menuChoice);
328
329     if(menuID == 0)
330       return;
331
332     switch(menuID)
333     {
334       case mApple:
335         if(menuItem == iAbout)
336           SysBeep(10);
```

```
      else
      {
        GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
        daDriverRefNum = OpenDeskAcc(itemName);
      }
      break;

    case mFile:
      if(menuItem == iQuit)
        gDone = true;
      break;
  }

  HiliteMenu(0);
}

// ########################################################################## doIncontent

void  doInContent(EventRecord *eventRecPtr,WindowPtr windowPtr)
{
  Point           mouseXY;
  ControlPartCode partCode;
  ControlHandle   controlHdl;

  mouseXY = eventRecPtr->where;
  GlobalToLocal(&mouseXY);

  if(partCode = FindControl(mouseXY,windowPtr,&controlHdl))
    doScrollBars(partCode,windowPtr,controlHdl,mouseXY);
}

// ########################################################################## doScrollBars

void  doScrollBars(ControlPartCode partCode,WindowPtr windowPtr,ControlHandle controlHdl,
                   Point mouseXY)
{
  DocRecHandle  docRecHdl;
  SInt16        oldControlValue;
  SInt16        scrollDistance;
  RgnHandle     updateRegion;

  docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));

  switch(partCode)
  {
    case kControlIndicatorPart:
      oldControlValue = GetControlValue(controlHdl);
      if(TrackControl(controlHdl,mouseXY,NULL))
      {
        scrollDistance = oldControlValue - GetControlValue(controlHdl);
        if(scrollDistance != 0)
        {
          if(controlHdl == (*docRecHdl)->hScrollbarHdl)
          {
            updateRegion = NewRgn();
            ScrollRect(&gPictRect,scrollDistance,0,updateRegion);
            InvalRgn(updateRegion);
            DisposeRgn(updateRegion);
          }
          else
          {
            // Vertical scroll bar scroll box handling here.
          }
        }
      }
      break;

    case kControlUpButtonPart:
    case kControlDownButtonPart:
    case kControlPageUpPart:
    case kControlPageDownPart:
      if(controlHdl == (*docRecHdl)->hScrollbarHdl)
        TrackControl(controlHdl,mouseXY,&actionProcedure);
      else
        // Vertical scroll via horizontal scrolling action procedure here.
      break;
  }
```

```
}

// ########################################################################## actionProcedure

pascal void  actionProcedure(ControlHandle controlHdl,ControlPartCode partCode)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
  SInt16        scrollDistance;
  SInt16        controlValue;
  RgnHandle     updateRegion;

  if(partCode)
  {
    windowPtr = (*controlHdl)->contrlOwner;
    docRecHdl = (DocRecHandle) (GetWRefCon(windowPtr));

    switch(partCode)
    {
      case kControlUpButtonPart:
      case kControlDownButtonPart:
        scrollDistance = 2;
        break;

      case kControlPageUpPart:
      case kControlPageDownPart:
        scrollDistance = (windowPtr->portRect.right - windowPtr->portRect.left) - 10;
        break;
    }

    if((partCode == kControlDownButtonPart) || (partCode == kControlPageDownPart))
      scrollDistance = -scrollDistance;

    controlValue = GetControlValue(controlHdl);
    if(((controlValue == GetControlMaximum(controlHdl)) && scrollDistance < 0) ||
       ((controlValue == GetControlMinimum(controlHdl)) && scrollDistance > 0))
      return;

    doMoveScrollBox(controlHdl,scrollDistance);

    updateRegion = NewRgn();
    ScrollRect(&gPictRect,scrollDistance,0,updateRegion);
    InvalRgn(updateRegion);
    DisposeRgn(updateRegion);

    if(scrollDistance == 2 || scrollDistance == -2)
      BeginUpdate(windowPtr);

    SetOrigin(GetControlValue((*docRecHdl)->hScrollbarHdl),0);
    DrawPicture(gPictureHdl,&gPictRect);
    SetOrigin(0,0);

    if(scrollDistance == 2 || scrollDistance == -2)
      EndUpdate(windowPtr);
  }
}

// ########################################################################## doMoveScrollBox

void doMoveScrollBox(ControlHandle controlHdl,SInt16 scrollDistance)
{
  SInt16    oldControlValue, controlValue, controlMax;

  oldControlValue = GetControlValue(controlHdl);
  controlMax = GetControlMaximum(controlHdl);

  controlValue = oldControlValue - scrollDistance;

  if(controlValue < 0)
    controlValue = 0;
  else if(controlValue > controlMax)
    controlValue = controlMax;

  SetControlValue(controlHdl,controlValue);
}

// ############################################################################################
```

Demonstration Program 2 Comments

When the program is run, the user should scroll the picture by dragging the scroll box, clicking in the scroll bar's gray areas, clicking in the scroll arrows and holding the mouse button down while the cursor is in the gray areas and scroll arrows.

Note that the picture which is scrolled in this demonstration is 600 pixels wide and 185 pixels high, that the window is 200 pixels wide by 200 pixels high, and that the 'CNTL' resource sets the control's maximum value to 400. Note also that the "SetOrigin" scrolling methodology is employed.

#define

Lines 40-48 establish constants relating to menu, window, picture, and control resources, menu IDs and menu item numbers. Line 49 defines MAXLONG as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

#typedef

Lines 53-58 define a data type for a document record. The single field of the document record will be assigned a handle to the control record for the vertical scroll bar.

Global Variables

gDone controls program termination. gInBackground has to do with foreground/background switching.

gPictRect is a Rect for the picture to be scrolled. gPictureHdl will be assigned a handle to the Picture structure associated with the 'PICT' resource.

main

main initialises the system software managers (Line 97), sets up the menus (Lines 101-111), opens a window and sets its graphics port as the current port for drawing (Lines 115-118), creates a relocatable block for the document record and assigns the handle to this block to the window record's refCon field (Lines 122-123), creates the control (Line 127), loads the 'PICT' resource (Line 131), and then enters the main event loop (Lines 135-141).

doGetControl

doGetControl creates the horizontal scroll bar. The call to GetNewControl (Line 170) allocates memory for the control record, inserts the control into the window's control list and draws the control. The handle to the control record is assigned to the appropriate field of the window's document record, which will maintain the association between the control and the window.

doGetPicture

doGetPicture loads the 'PICT' resource. At Line 177, the resource is loaded and a handle to the associated picture record is assigned to the global variable gPictureHdl. Line 179 copies the picture record's picFrame field to the global variable gPictRect. Lines 181-184 offset the rectangle so that the top and left fields are both set to 0.

doEvents

doEvents switches according to the type of event received.

doMouseDown

doMouseDown switches according to the window part code associated with a mouse-down event. Note that, in the case of a mouse-down in the content region, and if the program's window is the front window, the application-defined function doInContent is called.

doUpdate

doUpdate is called in response to update events.

Line 257 retrieves the handle to the window's document record. Line 263 ensures that the window's graphics port as the current port for drawing. Line 264 redraws the control if it intersects the window's visible region (which, between the BeginUpdate and EndUpdate calls, equates to the update region as it was before it was cleared by BeginUpdate).

Line 266 sets the window origin to the current scroll position, that is, to the position represented by the control's current value, ensuring that the correct part of the picture will be drawn by Line 267. Line 268 resets the window's origin to (0,0).

doActivate, doActivateWindow and doOSEvent

doActivate, doActivateWindow and doOSEvent are identical in purpose to those functions of the same name in the demonstration program Controls1.c

doMenuChoice

doMenuChoice handles menu choices from the Apple and File menus.

doInContent

doInContent establishes whether an inContent click was in a control.

Lines 361-362 extract the mouse-down coordinates from the where field of the event record and convert them to the local coordinates required by FindControl. If the call to FindControl at Line 364 returns a non-zero result, the mouse-down was in the control, in which case an application-defined scrollbar handling function is called (Line 365).

doScrollBars

doScrollBars receives the part code, the window pointer, the control's handle, and the mouse-down (local) coordinates, and performs the scrolling. The code is structured so that the handling of a vertical scroll bar, in addition to the horizontal scroll bar, could be readily included.

The handle to the window's document record is retrieved at Line 378.

Line 380 initiates a switch according to the received part code:

- If the mouse-down was in the scroll box (Line 382), the control's value at the time of the mouse-down is retrieved (Line 383). Control is then handed over to TrackControl (Line 384), which tracks user actions while the mouse button remains down. If the user releases the mouse button with the cursor inside the control box, the scroll distance (in pixels) is calculated by subtracting the control's value prior to the scroll from its current value (Line 386). If the user moved the scroll box (Line 387), and if this movement was to the horizontal scroll bar's scroll box (Line 389), the picture's pixels are scrolled by the specified scroll distance in the appropriate direction (Line 392), and the "vacated" area of the window following the scroll is added to the window's update region (Line 393). Note that the handling of the scroll box in a vertical scroll bar, if one existed, would be located at Line 398.

- If the mouse-down was in a scroll arrow or gray area (Lines 404-407), more specifically in the one of the horizontal scroll bar's scroll arrows or gray areas (Line 408), TrackControl takes control (Line 409) until the user releases the mouse button. This call to TrackControl, however, differs from that at Line 384 in one key respect: the third parameter contains a pointer to an action procedure. When a pointer to an action procedure is passed as the third parameter, TrackControl:

  - Repeatedly calls the action procedure while the mouse button remains down.

  - Passes the action procedure (1) a handle to the control and (2) the control's part code.

  (As an alternative to passing a pointer to the action procedure as a parameter in the TrackControl call, SetControlAction can be used to store a pointer to the action procedure in the contrlAction field in the control record. When (ControlActionUPP) -1, instead of a procedure pointer, is passed to TrackControl, TrackControl uses the action procedure pointed to in the control record.)

> ACTION PROCEDURES
>
> Action procedures (sometimes called hook procedures or call-back routines) refer to the ability of a system routine to call an application-defined function during its execution, thus extending the features of the routine. For source code that is to

be compiled as 680X0 code, but not as native PowerPC code, installing an action procedure simply involves passing a function pointer (that is, the address of the function) as an argument to the system routine.

Toolbox calls use Pascal calling conventions, and C and Pascal are different in their conventions. An action procedure can be written in C; however, in order to account for the difference in calling conventions, it must be declared using the pascal keyword.

PROCEDURE POINTERS AND UNIVERSAL PROCEDURE POINTERS

This call to TrackControl, incidentally, is your first encounter with one of the principal changes introduced by the Universal Headers.

Prior to the introduction of the Universal Headers, the prototype for TrackControl looked like this:

```
short  TrackControl(ControlHandle theControl,short thePoint,ProcPtr actionProc);
```

Indeed, you will still see it defined that way in Inside Macintosh and other references, such as THINK Reference. Notice that the third parameter is of type ProcPtr (procedure pointer). The third parameter is thus the address of a function, that is, an action procedure.

In the Universal Headers, the prototype for TrackControl looks like this:

```
ControlPartCode  TrackControl(ControlRef theControl, Point thePoint,
                              ControlActionUPP actionProc)
```

Notice that the third parameter is now of type ControlActionUPP (universal procedure pointer). Universal procedure pointers will be explained at Chapter 23 — Porting to the Power Macintosh. For the first 22 Chapters of Macintosh C, however, you may simply assume that, when the 680x0 compiler looks at a parameter which the Universal Headers say should be of type ControlActionUPP (or, indeed, any other data type defined with a "UPP" as the last three characters), it thinks that it is looking at a parameter of type ProcPtr. This is why a 680x0 compiler will compile Line 409 without protest; the third parameter is the address of the action procedure, so it is perfectly happy.

You will see in Chapter 23 that source code relating to system software routines like TrackControl, which require a universal procedure pointer (UPP) as a parameter, must be modified if it is to be capable of being compiled by a compiler which produces PowerPC code (as well as a compiler which produces 680x0 code).

actionProcedure

actionProcedure is the action procedure called by TrackControl. Because it is repeatedly called by TrackControl while the mouse button remains down, the scrolling it performs continues repeatedly until the mouse button is released, provided the cursor remains within the scroll arrow or gray area.

Firstly, if the cursor is not still inside the scroll arrow or gray area (Line 433), the action procedure exits. The following occurs only when the cursor is within the control.

At Lines 428-429, the pointer to the window record for the window which "owns" this control is retrieved from the control record's contrlOwner field, and the handle to the document record is retrieved from that window record's refCon field.

If the control part being used by the user to perform the scrolling is one of the scroll arrows, the distance to scroll (in pixels) is set to 2 (Lines 433-436). If the control part being used is one of the gray areas, the distance to scroll is set to the width of the window's content region minus 10 pixels (Lines 438-441). (Subtracting 10 pixels ensures that a small part of the pre-scroll display will appear at right or left (depending on the direction of scroll) of the post-scroll display.)

Lines 444-445 convert the distance to scroll to the required negative value if the user is scrolling towards the right.

Lines 447-450 defeat any further scrolling action if, firstly, the left scroll arrow is being used, the mouse button is still down and the document is at the minimum (left) scrolled position or, secondly, the right scroll arrow is being used, the mouse button is still down and the document is at the maximum (right) scrolled position.

Line 452 calls an application-defined function which adds/subtracts the distance to scroll to/from the control's current value and repositions the scroll box accordingly.

Lines 454-457 scroll the picture's pixels by the specified amount, and in the specified direction, as represented by the distance-to-scroll value. The "vacated" area is added to the window's update region at Line 456.

Lines 459-467 perform a update of the window's content region. If the scroll arrows are being used (Line 459) the call to BeginUpdate ensures that QuickDraw will redraw only the current two-pixel-wide update region. At Line 462, the SetOrigin call resets the window origin so that that part of the picture represented by the current scroll position is drawn. After the correct part of the picture is drawn, the window origin is reset to (0,0) (Line 464).

doMoveScrollBox

doMoveScrollBox is called from within the action procedure to reset the control's current value to reflect the scrolled distance, and to reposition the scroll box accordingly.

Line 477 retrieves the current control value. Line 478 retrieves the control's maximum value. Line 480 calculates the new control value by subtracting the received distance to scroll from the current control value. Lines 482-485 prevent the control's value from being set lower or higher than the control's minimum and maximum values respectively. The call to SetControlValue at Line 487 sets the new control value and repositions the scroll box.

Creating 'CNTL' Resources Using ResEdit

When learning to create the major resource types in ResEdit, it is recommended that you open Macintosh C to the page containing the relevant example resource definition in Rez input format and relate what you are doing within ResEdit to that definition. Accordingly, the methodology used in the following is to "walk through" selected 'CNTL' resources for the Controls1 demonstration program, relating what you see in ResEdit to the example definitions in this chapter.

Open the chap05cw_demo demonstration program folder and double-click on the Controls1.µ.rsrc icon to start ResEdit and open Controls1.µ.rsrc. The **Controls1.µ.rsrc** window opens.

Double-click the CNTL icon. The **CNTLs from Controls1.µ.rsrc** window opens. Several 'CNTL' resources (IDs 128 to 136) appear in the list. These are, in sequence, the 'CNTL' resources for:

- The pop-up menu (ID 128).
- Three radio buttons (IDs 129-131).
- Three check boxes (IDs 132-134).
- The vertical and horizontal scroll bars (IDs 135-136).

Radio Button Control

Double-click the list entry for ID = 129. The **CNTL ID = 129 from Controls1.µ.rsrc** window opens.

The following relates the example 'CNTL' resources for radio buttons in Rez input format in this chapter to the ResEdit display and interface:

`resource 'CNTL'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **CNTL** was clicked, and the dialog's **OK** button was clicked.
`(cDroplet,`	`cDroplet` is the 'CNTL' resource ID (129). Choose **Resource/Get Resource Info**. The **Info for CNTL 129 ...** window opens. Note the editable text item titled **ID:**. This is where you set the 'CNTL' resource ID. (ResEdit automatically assigns 128 as the 'CNTL' resource ID of the first 'CNTL' resource you create.) Note also that you can give the resource a name in this window. This is useful for identifying the various 'CNTL' resources in the **CNTL ID = 129 from Controls1.µ.rsrc** window.

preload, purgeable)	While the **Info for CNTL 129 ...** window is open, compare the **Attributes:** check boxes to the Resource Attributes table at Chapter 1. Note that both the **Purgeable** and the **Preload** checkboxes are checked. Close the **Info for CNTL 129 ...** window.
{13,23,31,142},	The control's rectangle. In the **CNTL ID = 129 ...** window, note the item **BoundsRect**. The sequence is top, left, bottom, right.
1,	Initial setting. Note the item **Value**.
visible,	Is control to be visible? Note the item **Visible** and the two related radio buttons.
1,	Maximum setting. Note the item **Max**.
0,	Minimum setting. Note the item **Min**.
radioButProc,	Control Definition ID. Note the item **ProcID**. (radioButProc = 2.)
0,	Reference constant. Note the item **RefCon**.
"Droplet"	Title of control. Note the item **Title**.

Close the **CNTL ID = 129 ...** window.

'CNTL' Resource For Pop-up Menu

In the **CNTLs from Controls1.rsrc** window, double-click the list entry for ID = 128. The **CNTL ID = 128 from Controls1.µ.rsrc** window opens.

The following relates the example 'CNTL' resources for a pop-up menu in Rez input format in this chapter to the ResEdit display and interface:

resource 'CNTL'	This was established when the resource was created by choosing Resource / Create New Resource. A small dialog opened, the item **CNTL** was clicked, and the dialog's **OK** button was clicked.
kPopUpCNTL,	kPopUpCNTL is the 'CNTL' resource ID (128). Choose **Resource/Get Resource Info**. The **Info for CNTL 128 ...** window opens. Note the editable text item titled **ID:**. This is where you set the 'CNTL' resource ID. (ResEdit automatically assigns 128 as the 'CNTL' resource ID of the first 'CNTL' resource you create and automatically increments the IDs for subsequently created 'CNTL' resources.)
preload, purgeable)	While the **Info for CNTL 128 ...** window is open, compare the **Attributes:** check boxes to the Resource Attributes table at Chapter 1. Note that both the **Purgeable** and **Preload** checkboxes are checked. Close the **Info for CNTL 129 ...** window.
{90,18,109,198},	The control's rectangle. In the **CNTL ID = 128 ...** window, note the item **BoundsRect**. The sequence is top, left, bottom, right.
popupTitle... ,	Title position. Note the item **Value**. 255 (0xFF) means popupTitleRightJust.
visible,	Is control to be visible? Note the item **Visible** and the two related radio buttons.
50,	Pixel width of title. Note the item **Max**.
kPopUpMenu,	'MENU' resource ID. Note the item **Min**.
popupMenuProc,	Control Definition ID. Note the item **ProcID**. (popupMenuProc = 1008.)
0,	Reference constant. Note the item **RefCon**.
"Speed"	Title of control. Note the item **Title**.

Close the **CNTLs from Controls1.µ.rsrc** window. Close the **Controls1.µ.rsrc** window without saving.