5

Version 1.2 (Frozen)

CONTROLS

Includes Demonstration Programs Controls1Pascal and Controls2Pascal

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.

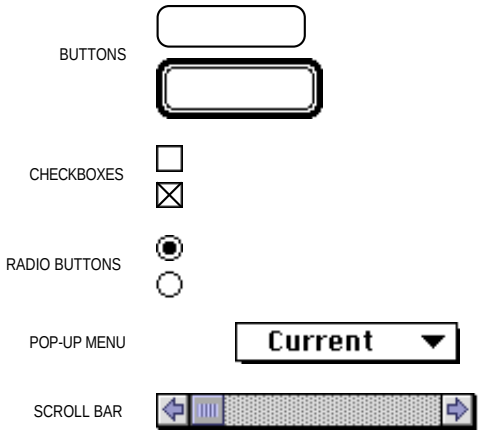

FIG 1 - STANDARD CONTROLS PROVIDED BY THE CONTROL MANAGER

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 `$0D` (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` in the Universal Interface file `controls.p`:

```
type
  ControlRecord = packed record
    nextControl:    ControlRef;
    contrlOwner:    WindowRef;
    contrlRect:     Rect;
    contrlVis:      UInt8;
    contrlHilite:   UInt8;
    contrlValue:    SInt16;
    contrlMin:      SInt16;
    contrlMax:      SInt16;
    contrlDefProc:  Handle;
    contrlData:     Handle;
    contrlAction:   ControlActionUPP;
    contrlRfCon:    SInt32;
    contrlTitle:    Str255;
  end;

  ControlPtr = ^ControlRecord;
  ControlHandle = ^ControlPtr;
  ControlRef = ControlHandle;
```

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.

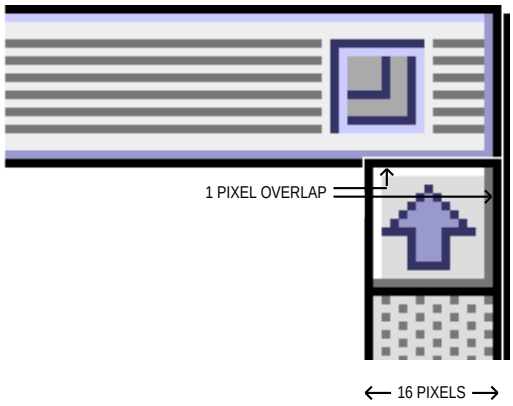

FIG 2 - CORRECT OVERLAP OF SCROLL BAR ON WINDOW FRAME

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.

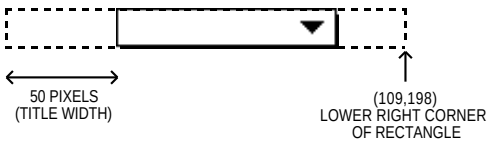

FIG 3 - DIMENSIONS OF SAMPLE POP-UP MENU

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	$0100
popupTitleItalic	$0200
popupTitleUnderline	$0400
popupTitleOutline	$0800
popupTitleShadow	$1000
popupTitleCondense	$2000
popupTitleExtend	$4000
popupTitleNoStyle	$0800
popupTitleLeftJust	$0000
popupTitleCenterJust	$0001
popupTitleRightJust	$00FF

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	$0001	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	$0004	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	$0008	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:
    - `NIL` 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.

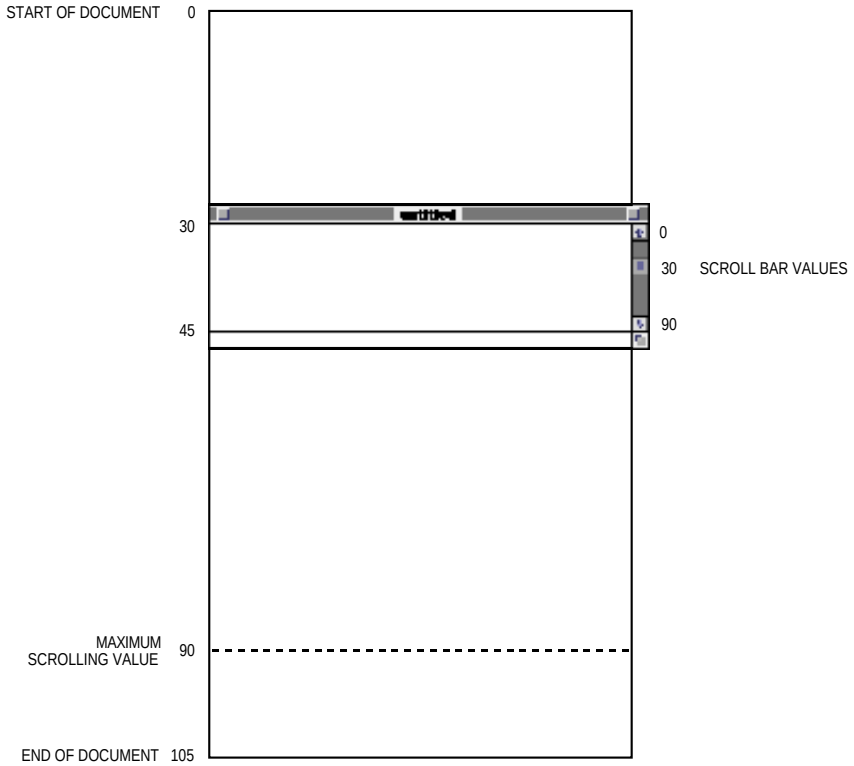

FIG 4 - SPATIAL RELATIONSHIP BETWEEN A DOCUMENT AND A WINDOW, AND THEIR REPRESENTATION IN A SCROLL BAR

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).

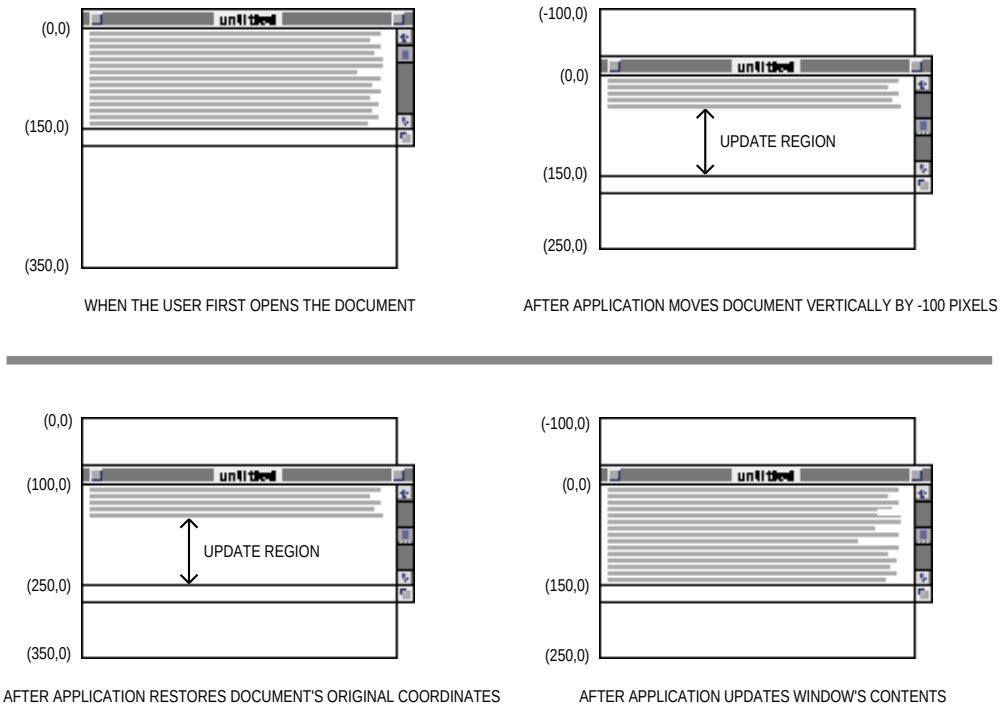

FIG 5 - SCROLLING A DOCUMENT IN A WINDOW

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 display
                             control title in the window font.
```

Pop-up Menu Variation Codes

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

Pop-up Title Characteristics

```
popupTitleBold       = 1 * (2**(8));
popupTitleItalic     = 1 * (2**(9));
popupTitleUnderline  = 1 * (2**(10));
popupTitleOutline    = 1 * (2**(11));
popupTitleShadow     = 1 * (2**(12));
popupTitleCondense   = 1 * (2**(13));
popupTitleExtend     = 1 * (2**(14));
popupTitleNoStyle    = 1 * (2**(15));

popupTitleLeftJust   = $00000000;
popupTitleCenterJust = $00000001;
popupTitleRightJust  = $000000FF;
```

Part Codes

```
inLabel              = 1
inMenu               = 2
inTriangle           = 4
inButton             = 10
inCheckBox           = 11
inUpButton           = 20
inDownButton         = 21
inPageUp             = 22
inPageDown           = 23
inThumb              = 129
```

Control Color Table Part Codes

```
cFrameColor          = 0
cBodyColor           = 1
cTextColor           = 2
cThumbColor          = 3
```

Data Types

```
typedef SInt16 ControlPartCode;
```

Control Record

```
  ControlRecord = packed record
    nextControl:    ControlRef;
    contrlOwner:    WindowRef;
    contrlRect:     Rect;
    contrlVis:      UInt8;
    contrlHilite:   UInt8;
    contrlValue:    SInt16;
    contrlMin:      SInt16;
    contrlMax:      SInt16;
    contrlDefProc:  Handle;
    contrlData:     Handle;
    contrlAction:   ControlActionUPP;
    contrlRfCon:    SInt32;
    contrlTitle:    Str255;
    end;

  ControlPtr = ^ControlRecord;
  ControlHandle = ^ControlPtr;
  ControlRef = ControlHandle;
```

Auxiliary Control Record

```
AuxCtlRec = record
     acNext:         Handle;
     acOwner:        ControlRef;
     acCTable:       CCTabHandle;
     acFlags:        SInt16;
     acReserved:     SInt32;
     acRefCon:       SInt32;
     end;

  AuxCtlPtr = ^AuxCtlRec;
  AuxCtlHandle = ^AuxCtlPtr;
```

Control Color Table Record

```
  CtlCTab = record
    ccSeed:         SInt32;
    ccRider:        SInt16;
    ctSize:         SInt16;
    ctTable:        ARRAY [0..3] OF ColorSpec;
    end;

  CCTabPtr = ^CtlCTab;
  CCTabHandle = ^CCTabPtr;
```

Routines

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

Creating Controls

```
function    NewControl(theWindow: WindowRef; var boundsRect: Rect; title: ConstStr255Param;
            visible: boolean; value: SInt16; min: SInt16; max: SInt16; procID: SInt16;
            refCon: SInt32): ControlRef;
function    GetNewControl(controlID: SInt16; owner: WindowRef): ControlRef;
```

Drawing Controls

```
procedure   ShowControl(theControl: ControlRef);
procedure   UpdateControls(theWindow: WindowRef; updateRegion: RgnHandle);
procedure   DrawControls(theWindow: WindowRef);
procedure   Draw1Control(theControl: ControlRef);
```

Handling Mouse Events in Controls

```
function    FindControl(thePoint: Point; theWindow: WindowRef; var theControl: ControlRef):
            ControlPartCode;
function    TrackControl(theControl: ControlRef; thePoint: Point;
            actionProc: ControlActionUPP): ControlPartCode;
function    TestControl(theControl: ControlRef; thePoint: Point): ControlPartCode;
```

Changing Control Settings and Display

```
procedure   SetControlValue(theControl: ControlRef; newValue: SInt16);
procedure   SetControlMinimum(theControl: ControlRef; newMinimum: SInt16);
procedure   SetControlMaximum(theControl: ControlRef; newMaximum: SInt16);
procedure   SetControlTitle(theControl: ControlRef; title: ConstStr255Param);
procedure   HideControl(theControl: ControlRef);
procedure   MoveControl(theControl: ControlRef; h: SInt16; v: SInt16);
procedure   SizeControl(theControl: ControlRef; w: SInt16; h: SInt16);
procedure   HiliteControl(theControl: ControlRef; hiliteState: ControlPartCode);
procedure   DragControl(theControl: ControlRef; startPoint: Point; var limitRect: Rect;
            var slopRect: Rect; axis: DragConstraint);
procedure   SetControlAction(theControl: ControlRef; actionProc: ControlActionUPP);
procedure   SetControlColor(theControl: ControlRef; newColorTable: CCTabHandle);
```

Determining Control Values

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

Removing Controls

```
procedure   DisposeControl(theControl: ControlRef);
procedure   KillControls(theWindow: WindowRef);
```

Demonstration Program 1

```
1   { ##########################################################################################
2   // Controls1Pascal.p
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  //
26  // •  An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit menus and a
27  //    pop-up menu (preload, non-purgeable).
28  //
29  // •  A 'WIND' resource (purgeable) (initially not visible).
30  //
31  // •  'CNTL' resources for the pop-up menu, radio buttons, checkboxes, button and
32  //    scroll bars (preload, purgeable) (initially visible).
```

```
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch
//    and is32BitCompatible flags set.
//
// ############################################################################### }

program Controls1Pascal(input, output);

{ ……… include the following Universal Interfaces }

uses

  Windows, Menus, Events, Types, Memory, Quickdraw, QuickdrawText, Fonts, Processes,
  TextUtils, Controls, OSUtils, TextEdit, Dialogs, ToolUtils, Devices, Segload, Sound;

{ ……… define the following constants }

const

 rMenubar = 128;
 rNewWindow = 128;

 mApple = 128;
    iAbout = 1;
 mFile = 129;
    iQuit = 11;
 mEdit = 130;

 cTimeZone = 128;
    pSydney = 1;
    pNewYork = 2;
    pLondon = 3;
    pRome = 4;

 cRed = 129;
 cWhite = 130;
 cBlue = 131;
 cShowgrid = 132;
 cShowrulers = 133;
 cButton = 134;
 cVScrollbar = 135;
 cHScrollbar = 136;

 kMaxLong = $7FFFFFFF;

{ ……… user-defined types }

type

DocRec = record
    popupControlHdl: ControlHandle;
    redHdl: ControlHandle;
    whiteHdl: ControlHandle;
    blueHdl: ControlHandle;
    showGridHdl: ControlHandle;
    showRulersHdl: ControlHandle;
    okButtonHdl: ControlHandle;
    vScrollbarHdl: ControlHandle;
    hScrollbarHdl: ControlHandle;
    end;

DocRecPointer = ^DocRec;
DocRecHandle = ^DocRecPointer;

{ ……… global variables }

var

gDone : boolean;
gInBackground : boolean;
menubarHdl : Handle;
menuHdl : MenuHandle;
myWindowPtr: WindowPtr;
docRecHdl : DocRecHandle;
eventRec : EventRecord;


```

```
{ ############################################################################ DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ############################################################################ DoMenuChoice }

procedure DoMenuChoice(menuChoice : longint);

  var
  menuID, menuItem : integer;
  itemName : string;
  daDriverRefNum : integer;

  begin
  menuID := HiWord(menuChoice);
  menuItem := LoWord(menuChoice);

  if (menuID = 0) then
    Exit(DoMenuChoice);

  case (menuID) of

    mApple:
      begin
      if (menuItem = iAbout)
        then  SysBeep(10)
        elsebegin
            GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
            daDriverRefNum := OpenDeskAcc(itemName);
            end;
      end;

    mFile:
      begin
      if (menuItem = iQuit) then
        gDone := true;
      end;

    end;
      {of case statement}

  HiliteMenu(0);
  end;
    {of procedure DoMenuChoice}

{ ######################################################################## DoPopupMenuChoice }

procedure DoPopupMenuChoice(controlValue : integer);

  begin
  case (controlValue) of

    pSydney:
      begin
      { Action as appropriate.}
      end;

    pNewYork:
      begin
      { Action as appropriate.}
      end;
```

```
187
188       pLondon:
189         begin
190         { Action as appropriate.}
191         end;
192
193       pRome:
194         begin
195         { Action as appropriate.}
196         end;
197       end;
198         {of case statement}
199
200     SysBeep(10);
201     end;
202       {of procedure DoPopupMenuChoice}
203
204   { ############################################################################ DoControls }
205
206   procedure DoControls(controlHdl : ControlHandle; docRecHdl : DocRecHandle);
207
208     begin
209     if ((controlHdl = docRecHdl^^.redHdl) or (controlHdl = docRecHdl^^.whiteHdl) or
210          (controlHdl = docRecHdl^^.blueHdl))
211
212       thenbegin
213         SetControlValue(docRecHdl^^.redHdl, 0);
214         SetControlValue(docRecHdl^^.whiteHdl, 0);
215         SetControlValue(docRecHdl^^.blueHdl, 0);
216         SetControlValue(controlHdl, 1);
217         end
218
219     elseif((controlHdl = docRecHdl^^.showGridHdl) or
220             (controlHdl = docRecHdl^^.showRulersHdl))
221
222       thenbegin
223         if (GetControlValue(controlHdl) = 1)
224           then SetControlValue(controlHdl, 0)
225         else SetControlValue(controlHdl, 1);
226         end
227
228     else  if ((controlHdl = docRecHdl^^.vScrollbarHdl) or
229               (controlHdl = docRecHdl^^.hScrollbarHdl))
230       then{Do scroll bars handling.}
231
232     else{Must be button.  Do button handling.};
233
234     SysBeep(10);
235     end;
236       {of procedure DoControls}
237
238   { ############################################################################ DoInContent }
239
240   procedure DoInContent(eventRec : EventRecord; myWindowPtr : WindowPtr);
241
242     var
243     controlHdl : ControlHandle;
244     controlValue : integer;
245     docRecHdl : DocRecHandle;
246     ignored : integer;
247
248     begin
249     GlobalToLocal(eventRec.where);
250
251     if (FindControl(eventRec.where, myWindowPtr, controlHdl) <> 0) then
252       begin
253       docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
254       if (controlHdl = docRecHdl^^.popupControlHdl)
255         thenbegin
256             ignored := TrackControl(controlHdl, eventRec.where, ControlActionUPP(-1));
257             controlValue := GetControlValue(controlHdl);
258             DoPopupMenuChoice(controlValue);
259             end
260
261         elseif (TrackControl(controlHdl, eventRec.where, nil) <> 0) then
262             DoControls(controlHdl, docRecHdl);
263       end;
```

```
  end;
    {of procedure DoInContent}

{ ############################################################################ DoAdjustScrollBars }

procedure DoAdjustScrollBars(myWindowPtr : WindowPtr);

  var
  winRect : Rect;
  docRecHdl : DocRecHandle;

  begin
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

  winRect := myWindowPtr^.portRect;

  HideControl(docRecHdl^^.vScrollbarHdl);
  HideControl(docRecHdl^^.hScrollbarHdl);

  MoveControl(docRecHdl^^.vScrollbarHdl, winRect.right - 15, winRect.top - 1);
  MoveControl(docRecHdl^^.hScrollbarHdl, winRect.left -1, winRect.bottom -15);

  SizeControl(docRecHdl^^.vScrollbarHdl, 16, winRect.bottom - 13);
  SizeControl(docRecHdl^^.hScrollbarHdl, winRect.right - 13, 16);

  ShowControl(docRecHdl^^.vScrollbarHdl);
  ShowControl(docRecHdl^^.hScrollbarHdl);

  DrawGrowIcon(myWindowPtr);
  end;
    {of procedure DoAdjustScrollBars}

{ ############################################################################ DoEraseGrowIcon }

procedure DoEraseGrowIcon(myWindowPtr : WindowPtr);

  var
  growBoxRect : Rect;

  begin
  SetPort(myWindowPtr);

  growBoxRect := myWindowPtr^.portRect;
  growBoxRect.left := growBoxRect.right - 15;
  growBoxRect.top := growBoxRect.bottom - 15;
  EraseRect(growBoxRect);
  end;
    {of procedure DoEraseGrowIcon}

{ ############################################################################ DoMouseDown }

procedure DoMouseDown(eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  partCode : integer;
  growRect : Rect;
  newSize : longint;

  begin
  partCode := FindWindow(eventRec.where, myWindowPtr);

  case (partCode) of

    inMenuBar:
      begin
      DoMenuChoice(MenuSelect(eventRec.where));
      end;

    inSysWindow:
      begin
      SystemClick(eventRec, myWindowPtr);
      end;

    inContent:
      begin
      if(myWindowPtr <> FrontWindow)
```

```
          thenSelectWindow(myWindowPtr)
          elseDoInContent(eventRec, myWindowPtr);
        end;

      inDrag:
        begin
        DragWindow(myWindowPtr, eventRec.where, qd.screenBits.bounds);
        end;

      inGoAway:
        begin
        if (TrackGoAway(myWindowPtr,eventRec.where)) then
          gDone := true;
        end;

      inGrow:
        begin
        growRect := qd.screenBits.bounds;
        growRect.top := 200;
        growRect.left := 275;
        newSize := GrowWindow(myWindowPtr, eventRec.where, growRect);
        if (newSize <> 0) then
          begin
          DoEraseGrowIcon(myWindowPtr);
          SizeWindow(myWindowPtr, LoWord(newSize), HiWord(newSize), true);
          DoAdjustScrollBars(myWindowPtr);
          end;
        end;

      inZoomIn, inZoomOut:
        begin
        if (TrackBox(myWindowPtr, eventRec.where, partCode)) then
          begin
          SetPort(myWindowPtr);
          EraseRect(myWindowPtr^.portRect);
          ZoomWindow(myWindowPtr, partCode, false);
          InvalRect(myWindowPtr^.portRect);
          DoAdjustScrollBars(myWindowPtr);
          end;
        end;
      end;
        {of case statement}
    end;
      {of procedure DoMouseDown}

{ ############################################################################## DoUpdate }

procedure DoUpdate(eventRec : EventRecord);

    var
    myWindowPtr : WindowPtr;

    begin
    myWindowPtr := WindowPtr(eventRec.message);

    BeginUpdate(myWindowPtr);

    if not (EmptyRgn(myWindowPtr^.visRgn)) then
      begin
      SetPort(myWindowPtr);
      UpdateControls(myWindowPtr, myWindowPtr^.visRgn);
      DrawGrowIcon(myWindowPtr);
      end;

    EndUpdate(myWindowPtr);
    end;
      {of procedure DoUpdate}

{ ######################################################################## DoActivateWindow }

procedure DoActivateWindow(myWindowPtr : WindowPtr; becomingActive : boolean);

    var
    docRecHdl : DocRecHandle;
    hiliteState : integer;

    begin
```

```
418     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
419
420     if (becomingActive)
421       then hiliteState := 0
422       else hiliteState := 255;
423
424     HiliteControl(docRecHdl^^.popupControlHdl, hiliteState);
425     HiliteControl(docRecHdl^^.redHdl,  hiliteState);
426     HiliteControl(docRecHdl^^.whiteHdl, hiliteState);
427     HiliteControl(docRecHdl^^.blueHdl, hiliteState);
428     HiliteControl(docRecHdl^^.showGridHdl, hiliteState);
429     HiliteControl(docRecHdl^^.showRulersHdl, hiliteState);
430     HiliteControl(docRecHdl^^.okButtonHdl, hiliteState);
431     HiliteControl(docRecHdl^^.vScrollbarHdl, hiliteState);
432     HiliteControl(docRecHdl^^.hScrollbarHdl, hiliteState);
433     end;
434       {of procedure DoActivateWindow}
435
436   { ############################################################################## DoActivate }
437
438   procedure DoActivate(eventRec : EventRecord);
439
440     var
441     myWindowPtr : WindowPtr;
442     becomingActive : boolean;
443
444     begin
445     myWindowPtr := WindowPtr(eventRec.message);
446
447     becomingActive := (BAnd(eventRec.modifiers, activeFlag) <> 0);
448
449     DoActivateWindow(myWindowPtr, becomingActive);
450     end;
451       {of procedure DoActivate}
452
453   { ############################################################################## DoOSEvent }
454
455   procedure DoOSEvent(eventRec : EventRecord);
456
457     begin
458     case BAnd(BSR(eventRec.message, 24), $000000FF) of
459
460       suspendResumeMessage:
461         begin
462         DrawGrowIcon(FrontWindow);
463         gInBackground := boolean(BAnd(eventRec.message, resumeFlag));
464         DoActivateWindow(FrontWindow, gInBackground);
465         end;
466
467       mouseMovedMessage:
468         begin
469         end;
470
471       end;
472         {of case statement}
473     end;
474       {of procedure DoOSEvent}
475
476   { ############################################################################## DoEvents }
477
478   procedure DoEvents(eventRec : EventRecord);
479
480     begin
481     case (eventRec.what) of
482       mouseDown:
483         begin
484         DoMouseDown(eventRec);
485         end;
486
487       updateEvt:
488         begin
489         DoUpdate(eventRec);
490         end;
491
492       activateEvt:
493         begin
494         DoActivate(eventRec);
```

```
        end;

      osEvt:
        begin
        DoOSEvent(eventRec);
        HiliteMenu(0);
        end;
      end;
        {of case statement}
    end;
      {of procedure DoEvents}

{ ############################################################################ DoGetControls }

procedure DoGetControls(myWindowPtr : WindowPtr);

  var
  docRecHdl : DocRecHandle;

  begin
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

  with docRecHdl^^ do
    begin
    popupControlHdl := GetNewControl(cTimeZone, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    redHdl := GetNewControl(cRed, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    whiteHdl := GetNewControl(cWhite, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    blueHdl := GetNewControl(cBlue, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    showGridHdl := GetNewControl(cShowgrid, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    showRulersHdl := GetNewControl(cShowrulers, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    okButtonHdl := GetNewControl(cButton, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    vScrollbarHdl := GetNewControl(cVScrollbar, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;

    hScrollbarHdl := GetNewControl(cHScrollbar, myWindowPtr);
    if (popupControlHdl = nil) then
      ExitToShell;
    end;
      {of with statement}

  end;
    {of procedure DoGetControls}

{ ################################################################## start of main program }

begin

  { ……initialise managers }

  DoInitManagers;

  { ……set up menu bar and menus }

  menubarHdl := GetNewMBar(rMenubar);
  if (menubarHdl = nil) then
```

```
572       ExitToShell;
573     SetMenuBar(menubarHdl);
574     DrawMenuBar;
575
576     menuHdl := GetMenuHandle(mApple);
577     if (menuHdl = nil)
578       then ExitToShell
579       else AppendResMenu(menuHdl,'DRVR');
580
581     { ……… open a window }
582
583     myWindowPtr := GetNewWindow(rNewWindow, nil, WindowPtr(-1));
584     if (myWindowPtr = nil) then
585       ExitToShell;
586     SetPort(myWindowPtr);
587
588     { ……… get block for document record, assign handle to window record refCon field }
589
590     docRecHdl := DocRecHandle(NewHandle(sizeof(DocRec)));
591     if (docRecHdl = nil) then
592       ExitToShell;
593
594     SetWRefCon(myWindowPtr, longint(docRecHdl));
595
596     { ……… get controls and show window }
597
598     DoGetControls(myWindowPtr);
599     DoAdjustScrollBars(myWindowPtr);
600     ShowWindow(myWindowPtr);
601
602     { ……… enter eventLoop }
603
604     gDone := false;
605
606     while not (gDone) do
607       if (WaitNextEvent(everyEvent, eventRec, kMaxLong, nil)) then
608         DoEvents(eventRec);
609
610   end.
611
612   { ########################################################################################## }
```

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 these 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.)

The const declaration block

Line 52-76 establish constants representing menu, window and control resource IDs, menu IDs and menu items. Line 76 defines kMaxLong as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

The type definition block

At Lines 80-95, 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.

The var declaration block

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.

The procedure DoMenuChoice

DoMenuChoice handles user choices from the drop-down menus.

The procedure DoPopUpMenuChoice

DoPopupMenuChoice branches according to the menu item number passed to it from Line 258.

The procedure 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 (Lines 209-210), the control values of all radio buttons are set to 0 before that for the selected control is set to 1 (Lines 213-216). If the control handle matches one of the checkboxes (Lines 219-220), the control value for that control is flipped (Lines 223-225). If the control handle matches that of one the scroll bars (Lines 228-229), appropriate scrolling handling is invoked (Line 230).

If a match has still not been found, the handle must be a match for the button's handle (Line 232, in which case the appropriate action is taken (Line 232).

The procedure DoInContent

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

Line 249 converts the mouse coordinates in the event record's where field to the local coordinates required in the call to FindControl at Line 251.

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 251 is compared with the handle to the pop-up control stored in the window's document record (Line 254). If they match, TrackControl is called (Line 256) with the procPtr field set to (-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 257). For a pop-up menu, this value represents the menu item number. At Line 258, the menu item number is passed to an application-defined procedure which handles the menu choice.

If the control handle returned by the FindControl call does not match the pop-up control's handle (Line 261), the handle must be to one of the other controls. In this case, TrackControl is called (Line 261), 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 procedure DoControls (Line 262).

The procedure DoAdjustScrollBars

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

At Line 276, a handle to the window's document record is retrieved from the window record's refCon field. At Line 278, 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 280-281 hide the scroll bars. The calls to MoveControl at Lines 283-284 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 286-287 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 289-290 show the scroll bars. Line 292 draws the grow icon.

The procedure DoEraseGrowIcon

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

The procedure DoMouseDown

DoMouseDown branches 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 340), and since all of the controls are located in the window's content region, a call to the application-defined procedure DoInContent is made at Line 342.

Lines 357-368 handle re-sizing of the window, which is of particular significance to the scroll bars. GrowWindow (Line 361) 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 362), an application-defined procedure is called to erase the grow box, SizeWindow is called to draw the window in its new size (Line 365), and an application-defined procedure is called at Line 366 to erase, move, resize and redraw the scroll bars.

Lines 371-380 handle window zooming, which is also of significance to the scroll bars. If the call to TrackBox at Line 372 returns a non-zero value, the window's content region is erased (Lines 374-375), ZoomWindow is called at Line 376 to redraw the window in its new state, InvalRect is called at Line 377 to add the entire content region to the update region, and an application-defined procedure is called at Line 378 to erase, move, resize and redraw the scroll bars.

The procedure 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 400), UpdateControls is called at Line 401 to draw those controls intersecting the current visible region, and DrawGrowIcon is called to draw the grow icon (Line 402).

The procedure 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 procedure DoOSEvent.)

At Line 417, a handle to the window's document record is retrieved from the window record's refCon field.

Lines 420-427 assign either 0 or 255 to the variable hiliteState depending on whether the window is becoming active or inactive. This variable is then used in the calls to HiliteControl at Lines 424-432 to either un-dim the controls and render them active or dim them and render them inactive.

The procedure DoActivate

DoActivate is called whenever the application receives an activate event for its window. At Line 447, 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 procedure DoActivateWindow at Line 449.

The procedure DoOSEvent

DoOSEvent handles operating system events.

If the event is a suspend or resume event (Line 460), DrawGrowIcon is called at Line 462 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 463). This variable is passed in the call to DoActivateWindow at Line 464. (Recall that the doesActivateOnFGSwitch flag is set in the 'SIZE' resource.)

The procedure DoEvents

DoEvents branches according to the event type reported.

The procedure DoGetControls

The DoGetControls procedure creates the controls from the various 'CNTL' resources. Firstly, at Line 515, the handle to the structure in which the handles to the control records will be stored is retrieved. Then, at Lines 517-554, 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.

The main program block

Within the main function, the system software managers are initialised (Line 566), the menu bar and drop-down menus are set up (Lines 570-579), and a zoomDocProc window is opened (Line 583).

At Line 590, a relocatable block the size of one document record is created. At Line 594, the handle to the block is assigned to the window record's refCon field.

At Line 598, a call is made to the application-defined function which creates the controls. Line 599 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 600 makes the window visible.

The main event loop is then entered (Lines 604-608).

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.

Demonstration Program 2

```
1   { ############################################################################################
2   // Controls2Pascal.p
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,
15  //    non-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  program Controls2Pascal(input, output);
29
30  { ……… include the following Universal Interfaces }
31
32  uses
33
34    Windows, Fonts, Quickdraw, Events, Types, Memory, Processes, Controls, Menus,
35    TextEdit, Dialogs, ToolUtils, OSUtils, Devices, Segload, Sound;
36
37  { ……… define the following constants }
38
39  const
40
41  rMenubar = 128;
42  rNewWindow = 128;
```

```
43  rPicture = 128;
44
45  mApple = 128;
46    iAbout = 1;
47  mFile = 129;
48    iQuit = 11;
49  mEdit = 130;
50
51  cHScrollbar = 128;
52
53  kMaxLong = $7FFFFFFF;
54
55  { ……… user-defined types }
56
57  type
58
59  DocRec = record
60    hScrollbarHdl : ControlHandle;
61    end;
62
63  DocRecHandle = ^^DocRec;
64
65  { ……… global variables }
66
67  var
68
69  gDone : boolean;
70  gInBackground : boolean;
71  gPictRect : Rect;
72  gPictureHdl : PicHandle;
73  menubarHdl : Handle;
74  menuHdl : MenuHandle;
75  myWindowPtr : WindowPtr;
76  docRecHdl : DocRecHandle;
77  eventRec : EventRecord;
78
79
80  { ########################################################################## DoInitManagers }
81
82  procedure DoInitManagers;
83
84    begin
85    MaxApplZone;
86    MoreMasters;
87
88    InitGraf(@qd.thePort);
89    InitFonts;
90    InitWindows;
91    InitMenus;
92    TEInit;
93    InitDialogs(nil);
94
95    InitCursor;
96    FlushEvents(everyEvent, 0);
97    end;
98      {of procedure DoInitManagers}
99
100 { ########################################################################## DoMoveScrollBox }
101
102 procedure DoMoveScrollBox(controlHdl : ControlHandle; scrollDistance : integer);
103
104   var
105   oldControlValue, controlValue, controlMax : integer;
106
107   begin
108   oldControlValue := GetControlValue(controlHdl);
109   controlMax := GetControlMaximum(controlHdl);
110
111   controlValue := oldControlValue - scrollDistance;
112
113   if (controlValue < 0)
114     thencontrolValue := 0
115     else  if (controlValue > controlMax)
116           then controlValue := controlMax;
117
118   SetControlValue(controlHdl, controlValue);
119   end;
```

```
    {of procedure DoMoveScrollBox}

{ ############################################################################ ActionProcedure }

procedure ActionProcedure(controlHdl : ControlHandle; partCode : ControlPartCode);

  var
  myWindowPtr : WindowPtr;
  docRecHdl : DocRecHandle;
  scrollDistance : integer;
  controlValue : integer;
  updateRegion : RgnHandle;

  begin
  if (partCode > 0) then
    begin
    myWindowPtr := controlHdl^^.contrlOwner;
    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

    case (partCode) of

      kControlUpButtonPart, kControlDownButtonPart:
        begin
        scrollDistance := 2;
        end;

      kControlPageUpPart, kControlPageDownPart:
        begin
        scrollDistance := (myWindowPtr^.portRect.right
                           - myWindowPtr^.portRect.left) - 10;
        end;
      end;
        {of case statement}

    if ((partCode = kControlDownButtonPart) or (partCode = kControlPageDownPart))
      then scrollDistance := -scrollDistance;

    controlValue := GetControlValue(controlHdl);
    if (((controlValue = GetControlMaximum(controlHdl)) and (scrollDistance < 0)) or
       ((controlValue = GetControlMinimum(controlHdl)) and (scrollDistance > 0)))
        then Exit(ActionProcedure);

    DoMoveScrollBox(controlHdl, scrollDistance);

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

    if((scrollDistance = 2) or (scrollDistance = -2)) then
      BeginUpdate(myWindowPtr);

    SetOrigin(GetControlValue(docRecHdl^^.hScrollbarHdl), 0);
    DrawPicture(gPictureHdl, gPictRect);
    SetOrigin(0, 0);

    if((scrollDistance = 2) or (scrollDistance = -2)) then
      EndUpdate(myWindowPtr);
    end;
  end;
    {of procedure ActionProcedure}

{ ############################################################################ DoScrollBars }

procedure DoScrollBars(partCode : ControlPartCode; myWindowPtr : WindowPtr;
            controlHdl : ControlHandle; mouseXY : Point);

  var
  docRecHdl : DocRecHandle;
  oldControlValue : integer;
  scrollDistance : integer;
  updateRegion : RgnHandle;
  ignored : integer;

  begin
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

```

```
197
198     case (partCode) of
199
200       kControlIndicatorPart:
201         begin
202         oldControlValue := GetControlValue(controlHdl);
203         if (TrackControl(controlHdl, mouseXY, nil) > 0) then
204           begin
205           scrollDistance := oldControlValue - GetControlValue(controlHdl);
206           if (scrollDistance <> 0) then
207             begin
208             if (controlHdl = docRecHdl^^.hScrollbarHdl)
209               thenbegin
210                   updateRegion := NewRgn;
211                   ScrollRect(gPictRect, scrollDistance, 0, updateRegion);
212                   InvalRgn(updateRegion);
213                   DisposeRgn(updateRegion);
214                   end
215
216               elsebegin
217                   {Vertical scroll bar scroll box handling here.}
218                   end;
219             end;
220           end;
221         end;
222
223       kControlUpButtonPart, kControlDownButtonPart, kControlPageUpPart,
224           kControlPageDownPart:
225
226         begin
227         if (controlHdl = docRecHdl^^.hScrollbarHdl)
228           thenignored := TrackControl(controlHdl, mouseXY, @ActionProcedure)
229           elsebegin
230                  {Vertical scroll via horizontal scrolling action procedure here.}
231               end;
232         end;
233
234       end;
235         {of case statement}
236
237     end;
238       {of procedure DoScrollBars}
239
240   { ############################################################################ DoIncontent }
241
242   procedure  DoInContent(eventRec : EventRecord; myWindowPtr : WindowPtr);
243
244     var
245     mouseXY : Point;
246     partCode : ControlPartCode;
247     controlHdl : ControlHandle;
248
249     begin
250     partCode := 0;
251     mouseXY := eventRec.where;
252     GlobalToLocal(mouseXY);
253
254     partCode := FindControl(mouseXY, myWindowPtr, controlHdl);
255     if (partCode <> 0) then
256       DoScrollBars(partCode, myWindowPtr, controlHdl, mouseXY);
257     end;
258       {of procedure DoInContent}
259
260   { ############################################################################## DoUpdate }
261
262   procedure  DoUpdate(eventRec : EventRecord);
263
264     var
265     myWindowPtr : WindowPtr;
266     docRecHdl : DocRecHandle;
267
268     begin
269     myWindowPtr := WindowPtr(eventRec.message);
270     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
271
272     BeginUpdate(myWindowPtr);
273
```

```
  if not (EmptyRgn(myWindowPtr^.visRgn)) then
    begin
    SetPort(myWindowPtr);
    UpdateControls(myWindowPtr, myWindowPtr^.visRgn);

    SetOrigin(GetControlValue(docRecHdl^^.hScrollbarHdl),0);
    DrawPicture(gPictureHdl, gPictRect);
    SetOrigin(0, 0);
    end;

  EndUpdate(myWindowPtr);
  end;
    {of procedure DoUpdate}

{ ############################################################################ DoActivateWindow }

procedure DoActivateWindow(myWindowPtr : WindowPtr; becomingActive : boolean);

  var
  docRecHdl : DocRecHandle;

  begin
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

  if (becomingActive)
    thenHiliteControl(docRecHdl^^.hScrollbarHdl, 0)
    elseHiliteControl(docRecHdl^^.hScrollbarHdl, 255);

  end;
    {of procedure DoActivateWindow}

{ ############################################################################## DoActivate }

procedure DoActivate(eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  becomingActive : boolean;

  begin
  myWindowPtr := WindowPtr(eventRec.message);

  becomingActive := (BAnd(eventRec.modifiers, activeFlag) <> 0);

  DoActivateWindow(myWindowPtr, becomingActive);
  end;
    {of procedure DoActivate}

{ ############################################################################### DoOSEvent }

procedure DoOSEvent(eventRec : EventRecord);

  begin
  case BAnd(BSR(eventRec.message, 24), $000000FF) of

    suspendResumeMessage:
      begin
      gInBackground := boolean(BAnd(eventRec.message, resumeFlag));
      DoActivateWindow(FrontWindow, gInBackground);
      end;

    mouseMovedMessage:
      begin
      end;

    end;
      {of case statement}
  end;
    {of procedure DoOSEvent}

{ ############################################################################ DoMenuChoice }

procedure DoMenuChoice(menuChoice : longint);

  var
  menuID, menuItem : integer;
  itemName : string;
```

```
351     daDriverRefNum : integer;
352
353     begin
354     menuID := HiWord(menuChoice);
355     menuItem := LoWord(menuChoice);
356
357     if (menuID = 0) then
358       Exit(DoMenuChoice);
359
360     case (menuID) of
361
362       mApple:
363         begin
364         if (menuItem = iAbout)
365           then  SysBeep(10)
366           elsebegin
367               GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
368               daDriverRefNum := OpenDeskAcc(itemName);
369               end;
370         end;
371
372       mFile:
373         begin
374         if (menuItem = iQuit) then
375           gDone := true;
376         end;
377
378       end;
379         {of case statement}
380
381     HiliteMenu(0);
382     end;
383       {of procedure DoMenuChoice}
384
385   { ############################################################################## DoMouseDown }
386
387   procedure DoMouseDown(eventRec : EventRecord);
388
389     var
390     myWindowPtr : WindowPtr;
391     partCode : integer;
392
393     begin
394     partCode := FindWindow(eventRec.where, myWindowPtr);
395
396     case (partCode) of
397
398       inMenuBar:
399         begin
400         DoMenuChoice(MenuSelect(eventRec.where));
401         end;
402
403       inSysWindow:
404         begin
405         SystemClick(eventRec, myWindowPtr);
406         end;
407
408       inContent:
409         begin
410         if(myWindowPtr <> FrontWindow)
411           thenSelectWindow(myWindowPtr)
412           elseDoInContent(eventRec, myWindowPtr);
413         end;
414
415       inDrag:
416         begin
417         DragWindow(myWindowPtr, eventRec.where, qd.screenBits.bounds);
418         end;
419
420       inGoAway:
421         begin
422         if (TrackGoAway(myWindowPtr,eventRec.where)) then
423           gDone := true;
424         end;
425
426       end;
427         {of case statement}
```

```
  end;
    {of procedure DoMouseDown}

{ ############################################################################## DoEvents }

procedure DoEvents(eventRec : EventRecord);

  begin
  case (eventRec.what) of
    mouseDown:
      begin
      DoMouseDown(eventRec);
      end;

    updateEvt:
      begin
      DoUpdate(eventRec);
      end;

    activateEvt:
      begin
      DoActivate(eventRec);
      end;

    osEvt:
      begin
      DoOSEvent(eventRec);
      HiliteMenu(0);
      end;
    end;
      {of case statement}
  end;
    {of procedure DoEvents}


{ ########################################################################## DoGetControl }

procedure DoGetControl(myWindowPtr : WindowPtr);

  var
  docRecHdl : DocRecHandle;

  begin
  docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));

  docRecHdl^^.hScrollbarHdl := GetNewControl(cHScrollbar, myWindowPtr);
  end;
    {of procedure DoGetControl}

{ ########################################################################## DoGetPicture }

procedure  DoGetPicture;

  begin
  gPictureHdl := GetPicture(rPicture);

  gPictRect := gPictureHdl^^.picFrame;
  gPictRect.right :=  gPictRect.right - gPictRect.left;
  gPictRect.left := 0;
  gPictRect.bottom := gPictRect.bottom - gPictRect.top;
  gPictRect.top := 0;
  end;
    {of procedure DoGetPicture}

{ ################################################################## start of main program }

begin

  { ……initialize managers }

  DoInitManagers;

  { ……set up menu bar and menus }

  menubarHdl := GetNewMBar(rMenubar);
  if (menubarHdl = nil) then
    ExitToShell;
```

```
505     SetMenuBar(menubarHdl);
506     DrawMenuBar;
507
508     menuHdl := GetMenuHandle(mApple);
509     if (menuHdl = nil)
510       thenExitToShell
511       elseAppendResMenu(menuHdl,'DRVR');
512
513     { ……… open a window }
514
515     myWindowPtr := GetNewWindow(rNewWindow, nil, WindowPtr(-1));
516     if (myWindowPtr = nil) then
517       ExitToShell;
518
519     SetPort(myWindowPtr);
520
521     { ……… get block for document record, assign handle to window record refCon field }
522
523     docRecHdl := DocRecHandle(NewHandle(sizeof(DocRec)));
524     SetWRefCon(myWindowPtr, longint(docRecHdl));
525
526     { ……… get controls }
527
528     DoGetControl(myWindowPtr);
529
530     { ……… get picture }
531
532     DoGetPicture;
533
534     { ……… enter eventLoop }
535
536     gDone := false;
537
538     while not (gDone) do
539       if (WaitNextEvent(everyEvent, eventRec, kMaxLong, nil)) then
540         DoEvents(eventRec);
541
542
543 end.
544
545 { ############################################################################### }
```

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.

The constant declaration block

Lines 41-53 establish constants relating to menu, window, picture, and control resources, menu IDs and menu item numbers. Line 53 defines kMaxLong as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

The type definition block

Lines 57-63 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.

The variable declaration block

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.

The procedure 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 108 retrieves the current control value. Line 109 retrieves the control's maximum value. Line 111 calculates the new control value by subtracting the received distance to scroll from the current control value. Lines 113-116 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 118 sets the new control value and repositions the scroll box.

The procedure 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 134), execution drops through to the bottom of the if statement and the action procedure exits. The following occurs only when the cursor is within the control.

At Lines 136-137, 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 the 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 141-144). 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 146-150). (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 154-155 convert the distance to scroll to the required negative value if the user is scrolling to the right. Lines 157-160 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 162 calls an application-defined procedure which adds/subtracts the distance to scroll to/from the control's current value and repositions the scroll box accordingly.

Lines 164-167 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 166.

Lines 169-177 perform a update of the window's content region. If the scroll arrows are being used, the call to BeginUpdate ensures that QuickDraw redraws only the current two-pixel-wide update region. At Line 172, 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 174).

The procedure 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 195.

Line 198 initiates a branch according to the received part code:

- If the mouse-down was in the scroll box (Line 200), the control's value at the time of the mouse-down is retrieved (Line 202). Control is then handed over to TrackControl (Line 203), 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 205). If the user moved the scroll box (Line 206), and if this movement was to the horizontal scroll bar's scroll box (Line 208), the picture's pixels are scrolled by the specified scroll distance in the appropriate direction (Line 211), and the "vacated" area of the window following the scroll is added to the window's update region (Line 212). Note that the handling of the scroll box in a vertical scroll bar, if one existed, would be located at Line 217.

- If the mouse-down was in a scroll arrow or gray area (Lines 223-224), more specifically in the one of the horizontal scroll bar's scroll arrows or gray areas (Line 227), TrackControl takes control (Line 228) until the user releases the mouse button. This call to TrackControl, however, differs from that at Line 203 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 procedure (or sometimes 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 procedure pointer (that is, the address of the procedure) as an argument to the system routine.

> PROCEDURE POINTERS AND UNIVERSAL PROCEDURE POINTERS
>
> This call to TrackControl, incidentally, is your first encounter with one of the principle changes introduced by the Universal Interfaces.
>
> Prior to the introduction of the Universal Interfaces, the prototype for TrackControl looked like this:
>
> ```
> function TrackControl(theControl: ControlHandle; thePoint: Point;
>                                                   actionProc: ProcPtr): integer;
> ```
>
> 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 Interfaces, the prototype for TrackControl looks like this:
>
> ```
> function TrackControl(theControl: ControlRef; thePoint: Point;
>                                 actionProc: ControlActionUPP): ControlPartCode;
> ```
>
> 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 23 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 228 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).

The procedure DoInContent

DoInContent establishes whether an inContent click was in a control.

Lines 251-252 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 254 returns a non-zero result, the mouse-down was in the control, in which case an application-defined scrollbar handling procedure is called (Line 256).

The procedure DoUpdate

DoUpdate is called in response to update events.

Line 270 retrieves the handle to the window's document record. Line 276 ensures that the window's graphics port as the current port for drawing. Line 277 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 279 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 280. Line 281 resets the window's origin to (0,0).

The procedures DoActivateWindow, doActivate and DoOSEvent

DoActivateWindow, DoActivate and DoOSEvent are identical in purpose to those functions of the same name in the demonstration program Controls1Pascal.p

The procedure DoMenuChoice

DoMenuChoice handles menu choices from the Apple and File menus.

The procedure DoMouseDown

DoMouseDown branches 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 procedure DoInContent is called.

The procedure DoEvents

DoEvents branches according to the type of event received.

The procedure DoGetControl

DoGetControl creates the horizontal scroll bar. The call to GetNewControl (Line 473) 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.

The procedure DoGetPicture

DoGetPicture loads the 'PICT' resource. At Line 482, the resource is loaded and a handle to the associated picture record is assigned to the global variable gPictureHdl. Line 484 copies the picture record's picFrame field to the global variable gPictRect. Lines 485-488 offset the rectangle so that the top and left fields are both set to 0.

The main program block

The main block initialises the system software managers (Line 498), sets up the menus (Lines 502-511), opens a window and sets its graphics port as the current port for drawing (Lines 515-519), creates a relocatable block for the document record and assigns the handle to this block to the window record's refCon field (Lines 523-524), creates the control (Line 528), loads the 'PICT' resource (Line 532), and then enters the main event loop (Lines 536-540).

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 chap05pascal_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.