



INTRODUCTION TO CONTROLS

Includes Demonstration Programs Controls1 & Controls2

Introduction

Prior to the introduction of Mac OS 8 and the Appearance Manager, the system software provided for only a limited range of **controls** (specifically, buttons, checkboxes, radio buttons, pop-up menus, and scroll bars) and the creation and handling of these desktop objects was relatively simple and straightforward. Mac OS 8 and the Appearance Manager, however, ushered in a very wide range of additional controls, extended the capabilities of the old controls, and provided a generally richer control environment. The result is that the subject of controls is now considerably more involved; accordingly, this chapter constitutes an introduction to controls only and addresses only the more basic controls — in the main, those control types that existed prior to the introduction of Mac OS 8 and the Appearance Manager. All but one of the remaining controls will be addressed at Chapter 14 — More on Controls. The remaining control will be addressed at Chapter 20 — Lists and Custom List Definition Functions.

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 dialog boxes and alert boxes. In this latter case, the Dialog Manager works with the Control Manager behind the scenes. The creation and handling of controls in dialog boxes and alert boxes will be addressed at Chapter 8 — Dialogs and Alerts. The creation and handling areas of this chapter are limited to the use of the Control Manager to create and handle controls in document windows.

Every control you create must be associated with a particular window. All the controls for a window are stored in a **control list**, a handle to which is stored in the `controlList` field of the window's window record.

Standard Controls

The term **standard controls** refers to controls whose control definition functions (see below) are provided by the system software. The term **custom controls** refers to controls that you provide yourself via a custom control definition function.

Historical Note

Prior to the introduction of Mac OS 8 and the Appearance Manager. It was often necessary to provide your own custom controls (for example, slider controls). However, the large range of controls now provided by the system software means

that it is all but inconceivable that you will ever need to create a custom control. A further consideration is the matter of Appearance-compliance. All of the new and revised controls provided by the system are Appearance-compliant. A non-Appearance-compliant custom control will create visual disharmony on the desktop, a situation much to be avoided.

Available Appearance-Compliant Standard Controls

The complete range of available standard controls, all of which are Appearance-compliant, is as follows. (Those control types and variants which existed prior to the introduction of Mac OS 8 and the Appearance Manager, and which have been revised so as to be Appearance-compliant, appear on a white background. Those control types and variants introduced with Mac OS 8 and the Appearance Manager appear on a light gray background. Those additional types and variants introduced with Mac OS 8.5 appear on a dark gray background.)

Control	Description	Variants
Push button	A control that appears on the screen as a rounded rectangle with a title centered inside. When the user clicks a push button, the application performs the action described by the button's title. Button actions are usually performed instantaneously. Examples include completing operations defined by a dialog box and acknowledging an error message in an alert box.	With title only. With colour icon to left of title. With colour icon to right of title.
Checkbox	A control that appears onscreen as a small square with an accompanying title. A checkbox displays one of three settings: on (indicated by a checkmark inside the box), off, or mixed (indicated by a dash inside the box).	(Non-auto-toggling. Auto-toggling.
Radio button	A control that appears onscreen as a small circle. A radio button displays one of three settings: on (indicated by a black dot inside the circle), off, or mixed (indicated by a dash inside the circle). A radio button is always a part of a group of related radio buttons in which only one button can be on at a time.	Non-auto-toggling. Auto-toggling.
Scroll bar	A control with which the user can change the portion of a document displayed within a window. A scroll bar is a light gray rectangle with scroll arrows at each end. Windows can have a horizontal scroll bar, a vertical scroll bar, or both. A vertical scroll bar lies along the right side of a window. A horizontal scroll bar runs along the bottom of a window. Inside the scroll bar is a rectangle called the scroll box . The rest of the scroll bar is called the gray area . The user can move through a document by manipulating the parts of the scroll bar.	Without live feedback. With live feedback.
Pop-up menu button	A control that is used to display a menu elsewhere than in the menu bar.	Fixed width. Variable width. Add resource. Use window font.
Bevel button	A button containing a self-descriptive icon, picture, text, or any combination of the three, that performs an action when pressed.	With small bevel. With normal bevel. With large bevel. The above with a pop-up menu either to the right or below.

Slider	A control that displays a range of values, magnitudes, or positions. A horizontally- and vertically-mobile indicator is used to increase or decrease the value.	Without live feedback. With live feedback. With tick marks. With directional indicator. With non-directional indicator.
Disclosure triangle	A triangular control governing how items are displayed in a list. The disclosure triangle can point in two directions: right and down. When the disclosure triangle points to the right, one item is displayed in the list. When the arrow points downward, the original item and its subitems are displayed in the list.	Right-facing. Left-facing. Right-facing, auto-tracking. Left-facing, auto-tracking.
Progress indicator	A control indicating that a lengthy operation is occurring. Two types of progress indicators can be used: an indeterminate progress indicator reveals that an operation is occurring but does not indicate its duration; a determinate progress indicator displays how much of the operation has been completed.	(One variant only. However, the progress indicator can be made determinate or non-determinate via a call to <code>SetControlData</code> .)
Little arrows	Up- and down-arrows accompanying a text box that contains a value, such as a date. Clicking the up arrow increases the value displayed. Clicking the down arrow decreases the value displayed.	(One variant only.)
Asynchronous (or chasing) arrows	A control which indicates through a simple animation that a background process is in progress.	(One variant only.)
Tab	A control that appears as a row of folder tabs on top of a pane. It allows multiple panes to appear in the same window.	Normal. Small.
Separator line	A control that draws a vertical or horizontal line used to visually separate groups of controls.	(One variant only.)
Primary group box	A control that consists of a rectangular two-pixel-wide frame which may or may not contain a title. It is used to provide a well-defined area in a dialog box into which text, pictures, icons or other controls can be embedded.	With text title. With checkbox title. With pop-up menu button title.
Secondary group box	A control that consists of a rectangular one-pixel-wide frame which may or may not contain a title. It is used to provide a well-defined area in a dialog box into which text, pictures, icons or other controls can be embedded.	With text title. With checkbox title. With pop-up button title.
Image well	A control that is used to display non-text visual content on a white background surrounded by a rectangular frame.	Without auto-tracking With auto-tracking.
Pop-up arrow	A control that simply draws the pop-up glyph.	Large, right-facing. Large, left-facing. Large, up-facing. Large, down-facing. Small, right-facing. Small, left-facing. Small, up-facing. Small, down-facing.
Placard	A rectangular control used as an information display.	(One variant only.)
Clock	A control that combines the features of little arrows and an edit text field into a control	Displays hours, minutes. Displays hours, minutes,

	which displays a date and/or time.	seconds. Displays date, month, year. Displays month, year.
User pane	A general purpose control which can be used as the root control for a window and as an embedder control in which other controls may be embedded. It can also be used to hook in callback functions for drawing, hit testing, etc.	(One variant only.)
Edit text	A control that appears as a rectangular box in which the user enters text to provide information to an application.	For windows. For dialog boxes. For passwords. Inline input.
Static text	A control that displays static (unchangeable by the user) text labels in a window.	(One variant only.)
Picture	A control used to display pictures.	Tracking. Non-tracking.
Icon	A control used to display icons.	Tracking. Non-tracking. Icon suite, tracking. Icon suite, non-tracking. All icon types, tracking. All icon types, non-tracking.
Window header	A control that runs along the top of a window's content region and provides information about the window's contents.	Window header. Window list view header.
List box	A control that combines a rectangular frame, scroll bar(s), and a scrolling list.	Non-autosizing. Autosizing.
Radio Group	A control that implements a radio button group.	(One variant only.)
Scrolling text box	A control that implements a scrolling text box.	Non-auto-scrolling. Auto-scrolling.

Definition of the Term "Controls"

Prior to the introduction of Mac OS 8 and the Appearance Manager, a control was defined as an "on-screen objects which the user can manipulate to cause an immediate action or to change settings to modify a future action". Given this previous definition, the question arises as to why such objects as, for example, separator lines and window headers are now implemented as controls. On the surface, it may appear that these objects are purely visual entities.

Part of the answer to that question has to do with the matter of Appearance-compliance. For example, using the provided separator line "control" to draw separator lines will ensure that those lines are drawn with the correct appearance, in both the activated and deactivated modes, regardless of which Appearance theme has been chosen by the user.

Another part of the answer has to do with the concept of **embedding** (see below). The window header control, for example, is not just the visual entity it might at first appear to be; it is actually a control in which other controls may be embedded. (As will be seen, the ability to embed other controls is a powerful new feature of some of the new controls.)

Since many of the new controls are not really controls "which the user can manipulate", a more accurate blanket definition might now be "any element of the user interface that is implemented by a control definition function" (see below).

Controls Addressed in This Chapter

Of the controls listed above, only those which might be termed the **basic controls** (push buttons, checkboxes, radio buttons, scroll bars, and pop-up menu buttons), together with **primary group boxes** (text title variant) and **user panes**, will be addressed in this chapter and its associated demonstration programs. These controls, with the exception of the user pane (which is invisible), are illustrated at Fig 1.

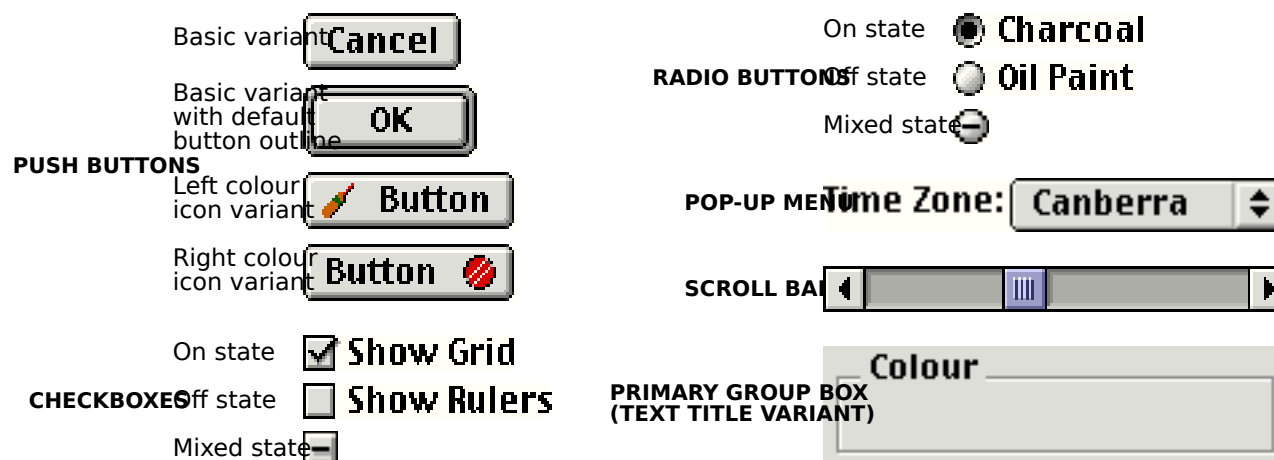


FIG 1 - THE BASIC CONTROLS AND THE PRIMARY GROUP BOX (TEXT TITLE VARIANT)

The Control Definition Function

A **control definition function** (CDEF) determines the appearance and behaviour of a control. Various Control Manager functions call a control's CDEF when they need to perform some control-related action. CDEFs are stored as resources of type 'CDEF'.

Just as a window definition function can describe variations of the same basic window, a CDEF 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 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 and variants addressed in this chapter and its associated demonstration programs, together with the derivation of those IDs. (Those variants introduced with Mac OS 8.5 are shown on a dark gray background.)

CDEF Resource ID	Variation Code	Control Definition ID (Value)	Control Definition ID (Constant)
23	0	$23 * 16 + 0 = 368$	kControlPushButtonProc
23	4	$23 * 16 + 4 = 374$	kControlPushButLeftIconProc
23	5	$23 * 16 + 5 = 375$	kControlPushButRightIconProc
23	1	$23 * 16 + 1 = 369$	kControlCheckBoxProc
23	3	$23 * 16 + 3 = 371$	kControlCheckBoxAutoToggleProc
23	2	$23 * 16 + 2 = 370$	kControlRadioButtonProc
23	4	$23 * 16 + 4 = 372$	kControlRadioAutoToggleButtonProc
24	0	$24 * 16 + 0 = 384$	kControlScrollBarProc
24	2	$24 * 16 + 2 = 386$	kControlScrollBarLiveProc
25	0	$25 * 16 + 0 = 400$	kControlPopupButtonProc
25	1	$25 * 16 + 1 = 401$	kControlPopupButtonProc + kControlPopupFixedWidthVariant
25	2	$25 * 16 + 2 = 402$	kControlPopupButtonProc + kControlPopupVariableWidthVariant
25	4	$25 * 16 + 4 = 404$	kControlPopupButtonProc + kControlPopupUseAddResMenuVariant
25	8	$25 * 16 + 8 = 408$	kControlPopupButtonProc + kControlPopupUseWFontVariant

10	0	$10 * 16 + 0$	= 160	kControlGroupBoxTextTitleProc
16	0	$16 * 16 + 0$	= 256	kControlUserPaneProc

Historical Note

With regard to the basic controls, these are the resource IDs and variation codes of the new Appearance-compliant CDEFs first issued with Mac OS 8 and the Appearance Manager. The old CDEFs have resource IDs of 0 (button, checkbox, radio button), 1 (scroll bar), and 63 (pop-up menu), and remain in the System file. The new definition functions are located in the Appearance extension.

Note that the one CDEF caters for both horizontal and vertical scroll bars. The CDEF determines whether a scroll bar is vertical or horizontal from the rectangle you specify when you create the control.

The Basic Controls, Primary Group Boxes (Text Title Variant), and User Panes

Push Buttons

You normally use push buttons in alert boxes and dialog boxes. Push 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 push buttons, you should designate one push button as the **default push button**, that is, the push button the user is most likely to click. (In modal and movable modal alert boxes, the Dialog Manager automatically outlines the default push button (see the default push button at Fig 1); however, your application must outline the push button in modeless dialog boxes.)

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

Non-Auto-Toggling Variant

When the non-auto-toggling variant is being used, and when the user clicks a checkbox in the **off state**, your application should call `SetControlValue` to set the control to the **on state** and place a checkmark in the box (see Fig 1). When the user again clicks the checkbox, your application should call `SetControlValue` to set the control to the off state and remove the checkmark from the box.

`SetControlValue` may also be used to place a dash in the box to indicate that the control is in the **mixed state** (see Fig 1). The mixed state is a special state which indicates that a selected range of items has some in the on state and some in the off state. For example, a text formatting checkbox for bold text would be in the mixed state if a text selection contained both bold and non-bold text.

Historical Note

The mixed state was introduced with Mac OS 8 and the Appearance Manager.

Auto-Toggling Variant

When the auto-toggling variant is being used, checkboxes automatically changes between their various states (on, off, and mixed) in response to user actions. Your application need only call the function `GetControl32BitValue` to get the checkbox's new state. There is no need to programmatically change the control's value after tracking successfully.

Historical Note

The auto-toggling variant was introduced with Mac OS 8.5.

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. 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.

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 button instead.

Non-Auto-Toggling Variant

When the non-auto-toggling variant is being used, and when the user clicks a radio button in the off state, your application should call `SetControlValue` to:

- Set that control to the **on state** and place a black dot in its circle (see Fig 1).
- Set the previously "on" button in the group to the **off state** and remove the black dot from its circle.

`SetControlValue` may also be used to place a dash in the circle to indicate that the control is in the **mixed state** (see Fig 1). The mixed state is a special state which shows that a selected range has a variety of items in the on state. For example, a set of radio buttons for selecting font size might have buttons representing 10- and 12-point sizes. If a passage of text with both 10- and 12-point text was selected, both the 10 and 12 buttons would appear in the mixed state.

Historical Note

The mixed state was introduced with Mac OS 8 and the Appearance Manager.

Auto-Toggling Variant

When the auto-toggling variant introduced with Mac OS 8.5 is being used, radio buttons automatically change between their various states (on, off, and mixed) in response to user actions. Your application need only call the function `GetControl32BitValue` to get the radio button's new state. There is no need to programmatically change the control's value after tracking successfully.

Historical Note

The auto-toggling variant was introduced with Mac OS 8.5.

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 **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.

As previously stated, the CDEF for scroll bars supports two variants. The only difference between the two variants is that one supports **live feedback** and the other does not. In the case of scroll bars, live feedback (a generic term) may be used to perform **live scrolling** of a document in a window. Live scrolling means that, when the user attempts to drag the scroll box, the scroll box moves and the document scrolls as the user moves the mouse. (Without live scrolling, only a ghosted image of the scroll box moves. In addition, the document is only scrolled, and the scroll box proper is only redrawn at its new location, when the user releases the mouse button.)

Historical Note

The live feedback capability was introduced with Mac OS 8 and the Appearance Manager.

Scroll Arrows and Gray Area

When the scroll arrows are clicked, your application should move the scroll box the appropriate distance in the direction of the arrow being clicked and scroll the window contents accordingly. Each click should move the window contents one unit in the chosen direction. (In a text document, a unit would typically be one line of text.)

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.

Scroll Box

Live Feedback Variant Not Used. When the live feedback variant is not being used, and when the user drags the scroll box, the Control Manager redraws the scroll box proper in its new position, and sets the control's value accordingly, when the user releases the mouse button. You must then ascertain the position (that is, the **value**) of the control box and, using this value, display the appropriate portion of the document.

Live Feedback Variant Used. When the live feedback variant is being used, and when the user drags the scroll box, the Control Manager continually redraws the scroll box, and continually returns the control's position (that is, its **value**) as the scroll box moves. Once again, your application uses this value to display the appropriate portion of the document.

Proportional Scroll Boxes

A proportional scroll box is one whose height (vertical scroll bars) or width (horizontal scroll bars) varies in relation to the height/width of the scroll bar so as to visually represent the proportion of the document visible in the window.

Under Mac OS 8.5 and later, if the user selects Smart Scrolling on in the Options tab in the Appearance control panel, your application's scroll boxes will appear as proportional

scroll boxes provided that you pass the size of the view area, in whatever units the scroll bar uses, to the function `SetControlViewSize`. The system automatically handles resizing the scroll box once your application supplies this information.

Historical Note

Proportional scroll boxes were introduced with Mac OS 8.5.

The following functions, introduced with Mac OS 8.5, are relevant to proportional scroll boxes:

Function	Description
<code>GetControlViewSize</code>	Obtains the size of the content to which a control's size is proportioned.
<code>SetControlViewSize</code>	Informs the Control Manager of the size of the content to which a control's size is proportioned.

Pop-Up Menu Buttons

Pop-up menu buttons provide the user with a simple way to choose from a list of choices. As an alternative to a group of radio buttons, a pop-up menu button 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 button's menu should be mutually exclusive.

Primary Group Boxes (Text Title Variant)

A primary group box (text title variant) is a control that consists of a rectangular two-pixel-wide frame which may or may not contain a **title**. Group boxes are used to associate, isolate, and distinguish groups of related controls, such as a group of radio buttons.

The primary group box is an **embedder** control (see below), meaning that you can embed other controls, such as radio buttons, checkboxes, and pop-up menu buttons, within it.

User Panes

The user pane is unique amongst the family of controls in that it has no visual representation. It has two main uses:

- Like the primary group box, it can be used as an embedder control, that is, other controls may be embedded within it.
- It provides a way to hook in application-defined functions, known as **user pane functions**, which perform actions such as drawing, hit testing, etc.

Activating, Deactivating, Hiding, and Showing Controls

Activating and Deactivating 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 dimmed state which indicates their inactive status to the user. The Control Manager displays inactive basic controls and inactive primary group boxes as shown at Fig 2.

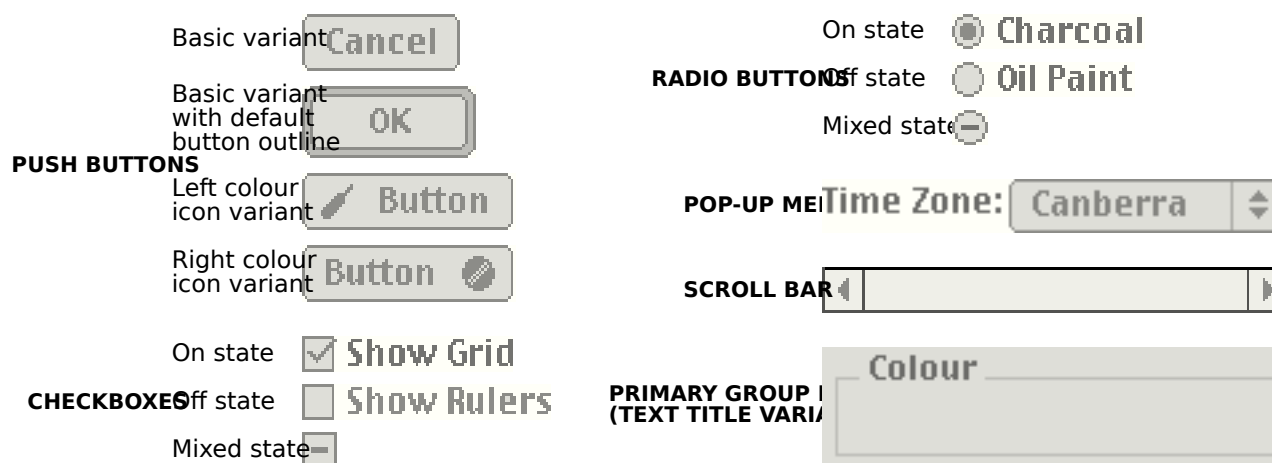


FIG 2 - THE BASIC CONTROLS AND THE PRIMARY GROUP BOX (TEXT TITLE VARIANT) IN

Activating and Deactivating Controls Other Than Scroll Bars

You use `ActivateControl` and `DeactivateControl` to make push buttons, checkboxes, radio buttons, pop-up menu buttons and primary group boxes active and inactive. You should make these controls inactive when:

- They are not relevant to the current context.
- The window in which they reside is not the active window.

Your application can ascertain whether a control is currently active or inactive using `IsControlActive`.

Historical Note

Prior to Mac OS 8 and the Appearance Manager, the `HiliteControl` function was used for this purpose. `HiliteControl` has other uses for which it may still be used; however it should not now be used to activate and deactivate controls.

Activating and Deactivating Scroll Bars

Scroll bars become irrelevant to the current context when the document being displayed is smaller than the window in which it is being displayed. To make a scroll bar inactive in this case, you typically use `SetControlMaximum` to make the scroll bar's maximum value (see below) equal to its minimum value (see below), 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` may be used to hide a control. `HideControl` erases a control by filling its enclosing rectangle with the owning window's background pattern.

`ShowControl` may be used to show a control. `ShowControl` makes the control visible and immediately draws the control within its window without using your window's standard updating mechanism.

`SetControlVisibility` may be used to both hide and show a control. With regard to showing a control, this function differs from `ShowControl` in that the option is available to "show" the control without redrawing it immediately.

Your application can ascertain whether a control is currently hidden or visible using `IsControlVisible`.

Historical Note

`SetControlVisibility` and `IsControlVisible` were introduced with Mac OS 8 and the Appearance Manager.

Visual Feedback From the Basic Controls

In response to a mouse-down event in a basic control, your application should call either `TrackControl` or `HandleControlClick`. These functions provide visual feedback when a mouse-down occurs in an active control by:

- Displaying push buttons, checkboxes and radio buttons in their pressed mode.
- Displaying and highlighting the items in pop-up menu buttons.
- Highlighting the scroll arrows in scroll bars.
- Moving the scroll box (live feedback variant of the scroll bar CDEF being used) or moving a ghosted image of the scroll box (live feedback variant not being used) when the user drags it.

Historical Note

`HandleControlClick` was introduced with Mac OS 8 and the Appearance Manager. It is identical to `TrackControl` except that it allows modifier keys to be passed in its third parameter. Some of the new controls, such as edit text fields and list boxes, require the ability for modifier keys to be passed in. If you use `HandleControlClick` with controls for which modifier keys are irrelevant, simply pass 0 in the `inModifiers` parameter.

Embedding Controls

In addition to providing a large number of additional control types, Mac OS 8 and the Appearance Manager introduced a richer environment for controls. The innovation of specific relevance to this chapter is control **embedding**. As previously stated, this involves the embedding of a control (or, more usually, a group of controls) in another control.

The Root Control

To embed controls in a window, you must first create a **root control** for that window. The root control, which is implemented as a user pane control, is the **container** for all other window controls and is at the base of what is known as the **embedding hierarchy**. In document windows, you create the root control by calling `CreateRootControl`. The root control may be retrieved by calling `GetRootControl`.

Once you have created a root control, new controls will be automatically embedded in the root control when they are created. One advantage of such embedding is that, when you wish to activate and deactivate all of the window's controls on window activation and deactivation, you can do so by simply activating and deactivating the root control. (If the root control did not exist, you would have to activate and deactivate all of the window's controls individually.) You can also hide and show all of the window's control by simply hiding and showing the root control.

Other Embedders

Certain other controls also have embedding capability. One such **embedder** control is the primary group box. This means that you can embed, say, a group of radio buttons in a primary group box (which would, in turn, be already automatically embedded in the root control), an arrangement which is illustrated conceptually at Fig 3. By acting on the group box alone, you can then activate, deactivate, hide, show, and move all four controls as a group.

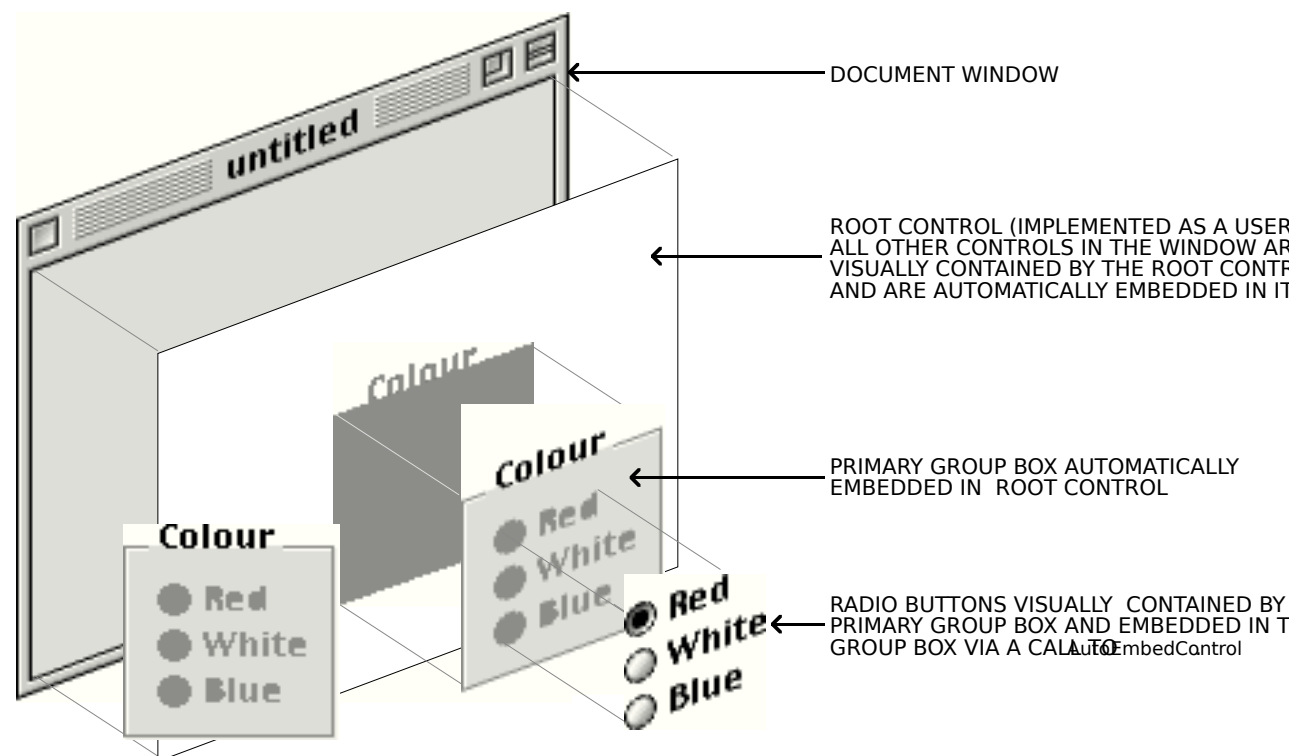


FIG 3 - THE ROOT CONTROL AND EMBEDDED CONTROLS

EmbedControl may be used to embed a control in another (embedder) control. However, where the control to be embedded is visually contained by the embedder, as is the case with the radio buttons in Fig 3, AutoEmbedControl would be more appropriate.

Other Advantages of Embedding

Drawing Order

As controls are created by your application, they are added to the head of the window's control list. When those controls are drawn in the absence of an embedding hierarchy, the Control Manager starts from the top of the control list, drawing the controls in the opposite order to the order in which they were created.

In the example at Fig 3, assume that there is no embedding hierarchy and that the radio buttons are created after the group box. This means that the group box will be drawn after the radio buttons, thus obscuring the radio buttons. An embedding hierarchy, however, enforces drawing order by drawing the embedding control before its embedded controls regardless of which is created first.

Hit Testing

Hit-testing is the process of testing whether a control is under the cursor at the time of a mouse-down event, and of identifying that control. For situations where controls are visually contained by other controls, an embedding hierarchy enforces orderly hit-testing by forcing an "inside-out" hit test aimed at determining the most deeply nested control that is hit by the mouse.

Latency

Latency pertains to the ability of the Control Manager to remember the activation and visibility status of an embedded control when its embedder is cycled between activated and deactivated, or between visible and hidden.

For example, assume that the radio button labelled White at Fig 3 has been separately deactivated by the application. When the primary group box is deactivated, the two remaining radio buttons will also be deactivated. When the primary group box is again enabled, the Control Manager remembers that the radio button labelled white was previously deactivated, and ensures that it remains in that mode.

Getting and Setting Control Data

Another refinement introduced with Mac OS 8 and the Appearance Manager was read and write access to the various attributes of a control. Essentially, this is a mechanism that allows the outside world to access a control's specialised data without exposing how that data is stored. It allows you to easily set and get control fonts, tell the push button CDEF to draw the default outline around a default push button, and many other useful things.

Each piece of information that a particular CDEF allows access to is referenced by a **tag**, which is a constant that is meaningful to the CDEF and which represents the data in question. Each tagged piece of data can be of any data type, such as a menu handle or a structure.

Control data tag constants are passed in the third parameter of the getter and setter routines `SetControlData` and `GetControlData`. The control data tag constants relevant to the basic controls are as follows:

Control Data Tag Constant	Meaning and Data Type Returned or Set
<code>kControlPushButtonDefaultTag</code>	Tells Appearance-compliant push buttons whether to draw a default ring, or returns whether the Appearance Manager has drawn a default ring, for the push button. Data type returned or set: Boolean
<code>kControlPushButtonCancelTag</code>	Gets or sets whether a given push button in a dialog or alert should be drawn with the appearance-specific adornments for a Cancel button. Data type returned or set: Boolean. Default is false.
<code>kControlPopupButtonMenuHandleTag</code>	Gets or sets the menu handle for a pop-up menu. Data type returned or set: MenuHandle
<code>kControlPopupButtonMenuIDTag</code>	Gets or sets the menu ID for a pop-up menu button. Data type returned or set: SInt16
<code>kControlPopupButtonExtraHeightTag</code>	Gets or sets the amount of extra white space in a pop-up menu button. Data type returned or set: SInt16. Default is 0.
<code>kControlGroupBoxTitleRectTag</code>	Get the rectangle that contains the title of a group box (and any associated control, such as a checkbox). Data type returned or set: Rect

Historical Notes

Calling `SetControlData` with the `kControlPushButtonDefaultTag` tag constant greatly simplifies the task of drawing the default ring around the default push button. Prior to Mac OS 8 and the Appearance Manager, your application had to provide its own function to do this.

kControlPushButtonCancelTag, kControlPopupButtonExtraHeightTag, and kControlGroupBoxTitleRectTag were introduced with Mac OS 8.5.

The Control Record

The Control Manager stores information about a control in a **control record**. A control record is defined by the data type ControlRecord:

```

TYPE
ControlRecord = PACKED RECORD
  nextControl:   ControlHandle;   { Next Control. }
  contrlOwner:   WindowPtr;       { Control's window. }
  contrlRect:    Rect;            { Rectangle. }
  contrlVis:     UInt8;           { 255 if visible, else 0. }
  contrlHilite:  UInt8;           { Highlight state. }
  contrlValue:   SInt16;          { Current setting. }
  contrlMin:     SInt16;          { Minimum Setting. }
  contrlMax:     SInt16;          { Maximum setting. }
  contrlDefProc: Handle;          { Control definition function. }
  contrlData:    Handle;          { Data used by contrlDefProc. }
  contrlAction:  ControlActionUPP; { Action function. }
  contrlRfCon:   SInt32;          { Reference constant. }
  contrlTitle:   Str255;          { Title. }
END;

ControlRecordPtr = ^ControlRecord;
ControlPtr = ^ControlRecord;
ControlHandle = ^ControlPtr;

```

Creating Your Application's Basic Controls

You typically create controls from resources of type 'CNTL', though you can create controls programmatically using the function NewControl.

'CNTL' Resources

When creating resources with Resorcerer, it is advisable that you refer to a diagram and description of the structure of the resource and relate that to the various items in the Resorcerer editing windows. Accordingly, the following describes the structure of the 'CNTL' resource.

Structure of a Compiled 'CNTL' Resource

Fig 4 shows the structure of a compiled 'CNTL' resource and how it "feeds" the control structure.

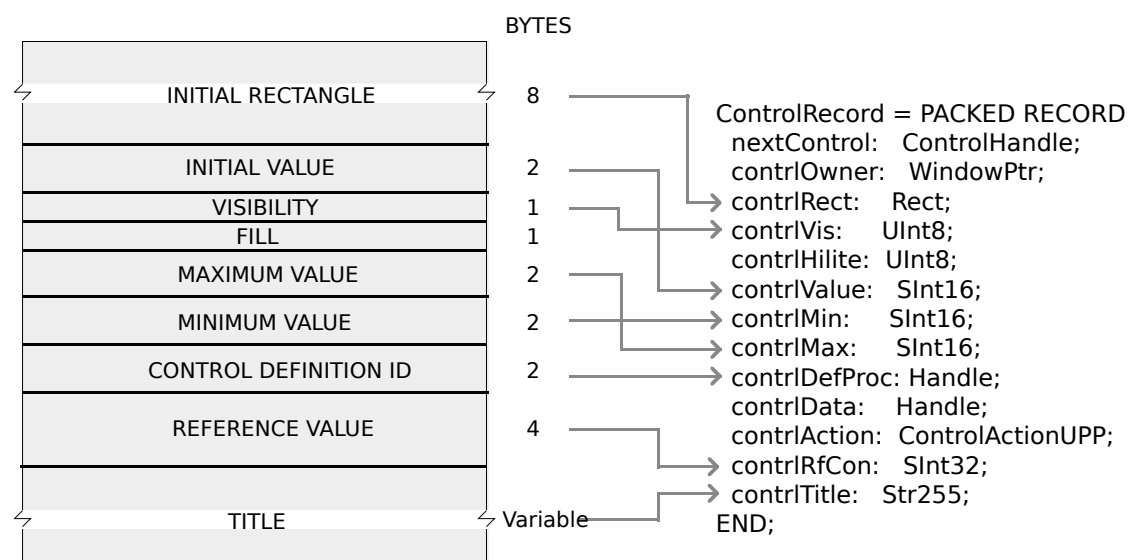


FIG 4 - STRUCTURE OF A COMPILED CONTROL ('CNTL') RESOURCE

The following describes the main fields of the 'CNTL' resource:

Field	Description
INITIAL RECTANGLE	The rectangle, specified in coordinates local to the window, that encloses the control and thus determines its size and location.
INITIAL VALUE	The initial value for the control. (See Values for Controls, below).
VISIBILITY	The visibility of the control. If this field contains the value <code>true</code> , <code>GetNewControl</code> draws the control immediately, without using the application's standard updating mechanism for windows. If this field contains <code>false</code> , the application may use <code>ShowControl</code> when it is prepared to display the control.
MAXIMUM VALUE	The maximum value of the control. (See Values for Controls, below).
MINIMUM VALUE	The minimum value for the control. (See Values for Controls, below).
CONTROL DEFINITION ID	The control definition ID, which the Control Manager uses to determine the CDEF for the control. (See The Control Definition Function, above).
REFERENCE VALUE	The control's reference value, which is set up and used only by the application (except when the control is the add resource variant of the pop-up menu button, in which case this field is used to specify the resource type).
TITLE	For controls that need a title, the string for that title. For controls that do not need or do not use titles, an empty string.

Values For Controls

The following lists the initial, minimum, and maximum value settings for the basic controls and the primary group box:

Control	Initial Value	Minimum Value	Maximum Value
Push button	0	0	1
Checkbox	0 (initially off), or 1 (initially on), or 2 (initially in mixed state).	0	1, or 2 if mixed state checkboxes are to be used.
Radio button	0 (initially off), or 1 (initially on), or 2 (initially in mixed state).	0	1, or 2 if mixed state radio buttons are to be used.
Scroll bar	Whatever initial value is appropriate (between the minimum and maximum settings).	Whatever minimum value is appropriate. (See Creating Scroll Bars, below.) The value must be between -32768 and	Whatever maximum value is appropriate. (See Creating Scroll Bars, below.) The value must be between -32768 and

		32768.	32768. When the maximum setting is equal to the minimum setting, the CDEF makes the scroll bar inactive. When the maximum setting is greater than the minimum setting, the CDEF makes the scroll bar active.
Pop-up menu button	A combination of values which instructs the Control Manager how to draw the control's title. (See Pop-up Menu Button Title Style Constants, below.)	Resource ID of the 'MENU' resource.	Width (in pixels) of the title. (See Pop-up Menu Button Title Width, below.)
Primary group box	Ignored if the group box is the text title variant.	Ignored if the group box is the text title variant.	Ignored if the group box is the text title variant.

Note that the title of each of the three value fields is somewhat of a misnomer in the case of the pop-up menu button.

Creating 'CNTL' Resources Using Resorcerer

Fig 5 shows a 'CNTL' resource being created with Resorcerer.

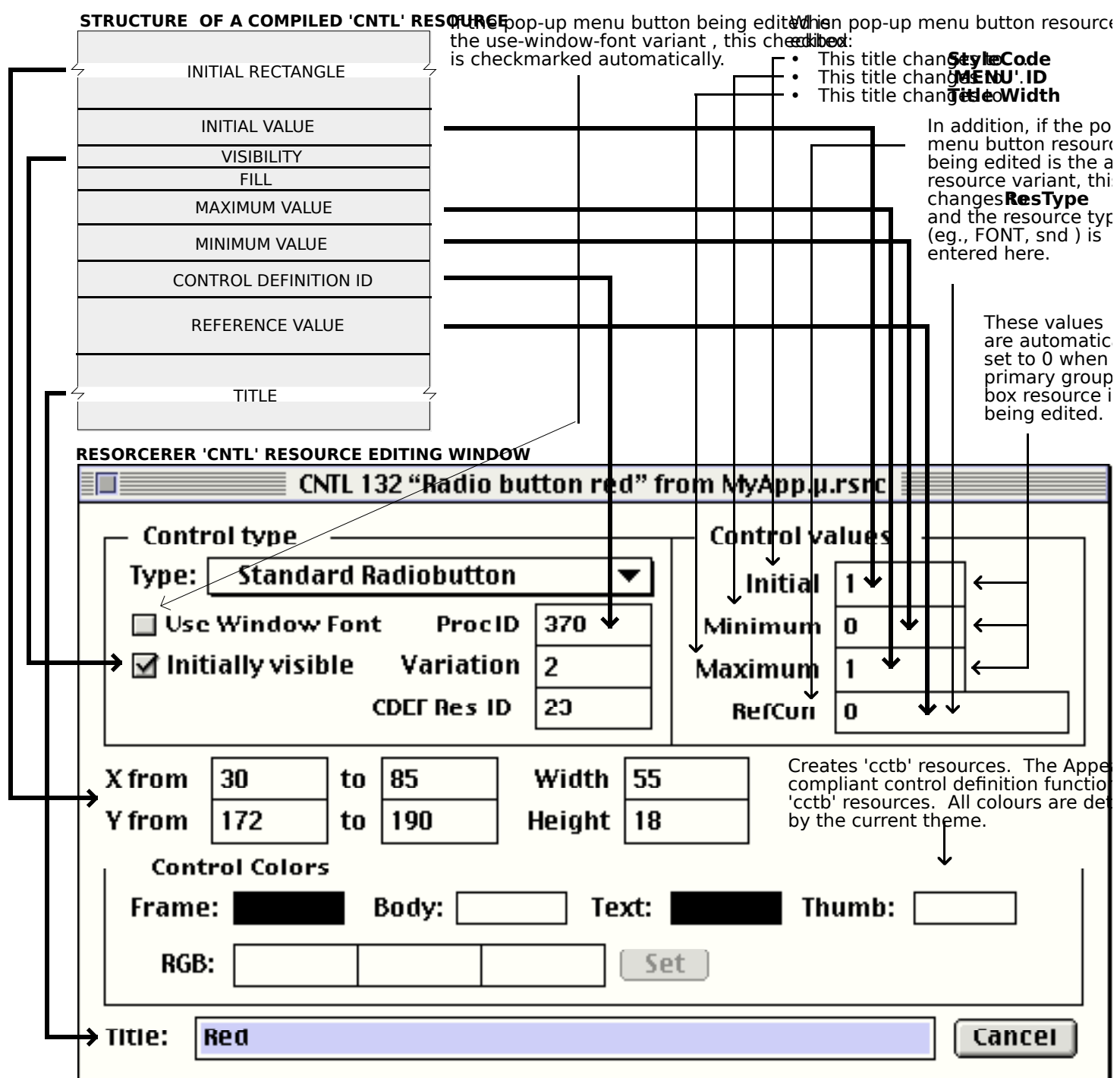


FIG 5 - CREATING A 'CNTL' RESOURCE USING RESORCERER

Creating Controls

GetNewControl and NewControl are used to create new controls in a document window. You typically use GetNewControl, which takes a 'CNTL' resource ID and a pointer to the window, creates a control structure from the information in the resource, adds the control structure to the control list for your window, and returns a handle to the control.

If the 'CNTL' resource specifies that a control is initially visible, the Control Manager uses the CDEF 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 push buttons, radio buttons, checkboxes or pop-up menu buttons in alert boxes or dialog boxes, Dialog Manager functions automatically use Control Manager functions to create the controls for you.

Creating Scroll Bars

The 'CNTL' resource for scroll bars should specify whether the live-feedback variant or the non-live-feedback variant is required. Within the 'CNTL' resource, you typically 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 value settings. (For example, for a window displaying a text document, you would typically calculate the number of lines of text and set the vertical scroll bar's maximum value according to the line count and the window's current height. You would set the control's value according to the part of the document to be initially displayed.) 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 for a document window 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 for a document window 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 6, the outside lines of the scroll bar should overlap the inside lines of the window frame.

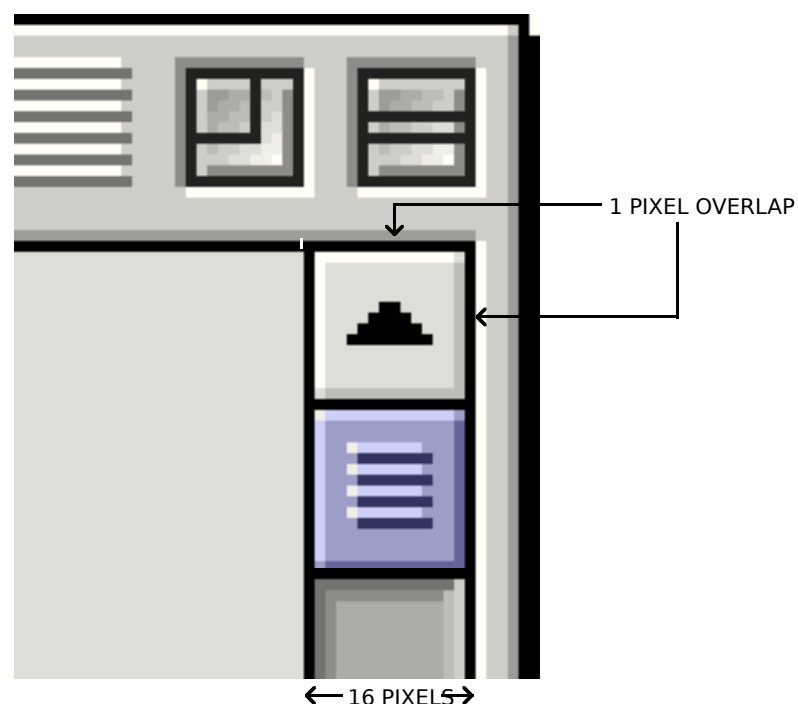


FIG 6 - CORRECT OVERLAP OF SCROLL BAR ON WINDOW

The following calculations¹ determine the rectangle for a vertical scroll bar for a document window:

Coordinate Calculation

¹ Do not include the title bar area in these calculations.

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 for a document window.

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.

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 an application-defined document structure associated with the window and then using Control Manager functions to change control settings.

Scroll Bars in Utility Windows. You can also use scroll bars in utility windows (see Fig 7). Size boxes in utility windows are 11-by-11 pixels, and the standard Appearance-compliant scroll bar can be made to fit into this space.

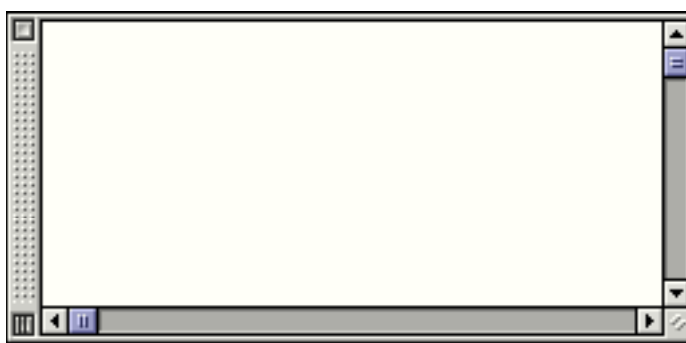


FIG 7 - SCROLL BARS IN A UTILITY WINDOW

Pop-up Menu Buttons

Pop-up Menu Button Title Style Constants

The constants and values for the initial value for pop-up menu buttons are as follows:

Constant	Value	Meaning
popupTitleBold	\$0100	Boldface font style
popupTitleItalic	\$0200	Italic font style
popupTitleUnderline	\$0400	Underline font style
popupTitleOutline	\$0800	Outline font style
popupTitleShadow	\$1000	Shadow font style
popupTitleCondense	\$2000	Condensed text
popupTitleExtend	\$4000	Extended text
popupTitleNoStyle	\$0800	Monostyle text

Pop-up Menu Button Title Width

Fig 8 shows the relationship between the title width and the enclosing rectangle of a pop-up menu box.

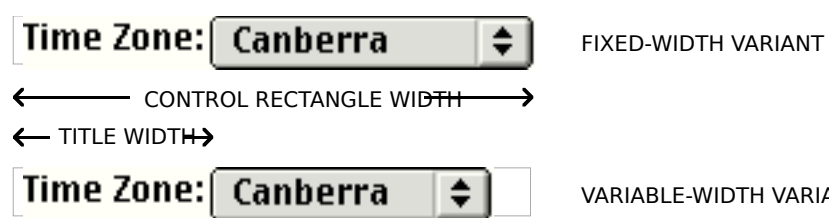


FIG 8 - POP-UP MENU BUTTON TITLE WIDTH AND CONTROL R

Menu Width Adjustment

If the base variant or the variable width variant is being used, and whenever the pop-up menu button is redrawn, the CDEF calls `CalcMenuSize` to calculate the size of the menu associated with the control. If the sum of the width of the title, the longest item in the menu, the arrows, and a small amount of "white space" is less than the width of the control rectangle, the width of the pop-up button will be reduced for drawing purposes (see Fig 7). If the calculated width is greater than the width of the control rectangle, the longer menu items will be truncated with an added ellipsis so that the drawn pop-up will not exceed the width of the control rectangle.

Menu Items and Control Values

When it creates the control, `GetNewControl` assigns the item number of the first menu item to the `controlValue` field of the control structure and sets the `controlMax` field to the number of items in the pop-up menu button. When the user chooses a different menu item, the Control Manager changes the `controlValue` field to that item number.

Adding Resource Names as Items

If the add resource variant of the pop-up menu button is being used, the Control Manager coerces the value in the control structure's `controlRfCon` field to the type `ResType` and then uses `AppendResMenu` to add items of that type. For example, if you specify `FONT` in the `ResType` item in the Resorcerer 'CNTL' resource editing window, the CDEF appends a list of fonts installed in the system to the menu specified at the 'MENU' ID item. Note that, after the control has been created, your application can use the `controlRfCon` field for whatever purpose it requires.

Setting the Font of a Control's Title

You can set the font of any control's title independently of the system font or window font.

Historical Note

Prior to Mac OS 8 and the Appearance Manager, your only choices were the system font and, if the control had a variant that used it (as did the pop-up menu control), the window font.

To set the font of a control's title, you pass a pointer to a **control font style structure** in the `inStyle` parameter of the function `SetControlFontStyle`.

Control Font Style Structure

```

TYPE
ControlFontStyleRec = RECORD
  flags:      SInt16;
  font:       SInt16;
  size:       SInt16;
  style:      SInt16;
  mode:       SInt16;
  just:       SInt16;
  foreColor:  RGBColor;
  backColor:  RGBColor;
END;

ControlFontStyleRecPtr = ^ControlFontStyleRec;
ControlFontStylePtr   = ^ControlFontStyleRec;

```

Field Descriptions

- flags** A signed 16-bit integer specifying which fields of the structure should be applied to the control (see Control Font Style Flag Constants, below). If none of the flags in the `flags` field of the structure are set, the control uses the system font unless the control variant `kControlUsesOwningWindowsFontVariant` has been specified, in which case the control uses the window font.
- font** If the `kControlUseFontMask` bit is set, this field will contain an integer indicating the ID of the font family to use. If this bit is not set, then the system default font is used. A meta font constant can be specified instead (see Meta Font Constants, below).
- size** If the `kControlUseSizeMask` bit is set, this field will contain an integer representing the point size of the text. If the `kControlAddSizeMask` bit is set, this value will represent the size to add to the current point size of the text. A meta font constant can be specified instead (see Meta Font Constants, below).
- style** If the `kControlUseStyleMask` bit is set, this field will contain an integer specifying which styles to apply to the text. If all bits are clear, the plain font style is used. The bit numbers and the styles they represent are as follows:
- | Bit Value | Style |
|-----------|-----------|
| 0 | Bold |
| 1 | Italic |
| 2 | Underline |
| 3 | Outline |
| 4 | Shadow |
| 5 | Condensed |
| 6 | Extended |
- mode** If the `kControlUseModeMask` bit is set, this field will contain an integer specifying how characters are drawn in the bit image. (See Chapter 12 — Drawing With QuickDraw for a discussion of transfer modes.)

- just** If the `kControlUseJustMask` bit is set, this field will contain an integer specifying text justification (left, right, centered, or system script direction).
- foreColor** If the `kControlUseForeColorMask` bit is set, this field will contain an RGB (red-green-blue) colour to use when drawing the text.
- backColor** If the `kControlUseBackColorMask` bit is set, this element will contain an RGB (red-green-blue) colour to use when drawing the background behind the text. (Note that, in certain text modes, background colour is ignored.)

Control Font Style Flag Constants

You can pass one or more of the following control font style flag constants in the `flags` field of the control font style structure to specify those fields of the structure that are to be applied to the control. (The constant shown on a dark gray background was introduced with Mac OS 8.5.) If none of the flags in the `flags` field are set, the control uses the system font unless a control with a variant that uses the window font has been specified.

Constant	Value	Meaning
<code>kControlUseFontMask</code>	0x0001	The <code>font</code> field of the control font style structure is applied to the control.
<code>kControlUseFaceMask</code>	0x0002	The <code>style</code> field of the control font style structure is applied to the control. This flag is ignored if you specify a meta font value (see <i>Meta Font Constants</i> , below).
<code>kControlUseSizeMask</code>	0x0004	The <code>size</code> field of the control font style structure is applied to the control. This flag is ignored if you specify a meta font value (see <i>Meta Font Constants</i> , below).
<code>kControlUseForeColorMask</code>	0x0008	The <code>foreColor</code> field of the control font style structure is applied to the control. This flag only applies to static text controls.
<code>kControlUseBackColorMask</code>	0x0010	The <code>backColor</code> field of the control font style structure is applied to the control. This flag only applies to static text controls.
<code>kControlUseModeMask</code>	0x0020	The text mode specified in the <code>mode</code> field of the control font style structure is applied to the control.
<code>kControlUseJustMask</code>	0x0040	The <code>just</code> field of the control font style structure is applied to the control.
<code>kControlUseAllMask</code>	0x00FF	All flags in this mask will be set except <code>kControlUseAddFontSizeMask</code> .
<code>kControlUseAddFontSizeMask</code>	0x0100	The Dialog Manager will add a specified font size to the <code>size</code> field of the control font style structure. This flag is ignored if you specify a meta font value (see <i>Meta Font Constants</i> , below).
<code>kControlAddToMetaFontMask</code>	0x0200	The control may use a meta font while also adding other attributes to the font.

Meta Font Constants

You can use the following meta font constants in the `font` field of the control font style structure to specify the style, size, and font family of a control's font. You should use these meta font constants whenever possible because the system font can change, depending upon the current theme. If none of these constants are specified, the control uses the system font unless a control with a variant that uses the window font has been specified.

Constant	Value	Meaning In Roman Script System
<code>kControlFontBigSystemFont</code>	\$0001	Use the system font. (For the Roman script system,

		this is Charcoal 12.)
kControlFontSmallSystemFont	\$0002	Use the small system font. (For the Roman script system, this is Geneva 10.)
kControlFontSmallBoldSystemFont	\$0004	Use the small emphasised system font. (For the Roman script system, this is Geneva 10.)

Another advantage of using these meta font constants is that you can be sure of getting the correct font on a Macintosh using a different script system, such as kanji.

Updating, Moving, and Removing Controls

Updating Controls

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

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

Moving Controls

You can change the position of a control using `MoveControl`, which erases the control, offsets the control's control rectangle, and redraws it at the specified new location

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 structure and associated data structures from memory. `KillControls` will dispose of all of a window's controls at once.

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.
- Call `TrackControl` OR `HandleControlClick` to handle user interaction with the control as long as the user holds the mouse button down. The `actionProc` parameter passed to `TrackControl`, or the `inAction` parameter passed to `HandleControlClick`, should be as follows:
 - nil for push buttons, checkboxes and radio buttons.
 - For scroll arrows and gray areas of scroll bars, a pointer which invokes an application-defined **action function** which, in turn, causes the document to scroll as long as the user holds the mouse button down.
 - For the scroll box of scroll bars:

- nil if the non-live-feedback variant is being used.
- If the live-feedback variant is being used, a pointer which invokes an application-defined **action function** which, in turn, causes the document to scroll while the scroll box is being dragged.
- ControlActionUPP(-1) for pop-up menu buttons. This causes TrackControl and HandleControlClick to use the action function defined within the pop-up CDEF, a pointer to which is stored in the contrlAction field of the control structure.

For source code that only needs to be compiled as 680x0 code, the pointer which invokes the application-defined action function need only be a procedure pointer, that is, the address of the action function. For source code that needs to be compiled as PowerPC code, or as both PowerPC code and 680x0 code, the pointer must be a Universal Procedure Pointer (UPP).

Note that, as an alternative to passing these pointers in the actionProc parameter of TrackControl, or the inAction parameter of HandleControlClick, you can preset the action function by passing the pointer in the actionProc parameter of SetControlAction. (Ordinarily, you would call SetControlAction immediately after the control is created. SetControlAction stores the pointer in the contrlAction field of the control structure.) In this case, you must pass ControlActionUPP(-1) in the actionProc and inAction parameters of TrackControl and HandleControlClick.

- When TrackControl or HandleControlClick 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 push button title if the cursor is over a push 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.
 - 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.
 - If the non-live-feedback scroll bar variant is being used, and when the cursor is over the scroll box, determine where the user has dragged the scroll box, and then display the corresponding portion of the document.
 - Use the new setting chosen by the user when the cursor is over a pop-up menu button.

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 **control 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 control part code.)

Historical Note

A new routine similar to FindControl was introduced with Mac OS 8 and the Appearance Manager. This new routine (FindControlUnderMouse) will return a handle to the control even if no part was hit and can determine whether a mouse-down event has occurred even if the control is deactivated, whereas FindControl will not.

A control part code is an integer from 1 to 255. Part codes are assigned to a control by its CDEF. The CDEFs for the basic controls define the following part codes:

Constant	Value	Meaning
kControlNoPart	0	Event did not occur in any control. Also unhighlights any highlighted part of the control when passed to the <code>HiLiteControl</code> function.
kControlLabelPart	1	Event occurred in the label of a pop-up menu button.
kControlMenuPart	2	Event occurred in the menu of a pop-up menu button.
kControlButtonPart	10	Event occurred in a push button.
kControlCheckBoxPart	11	Event occurred in a checkbox.
kControlRadioButtonPart	12	Event occurred in a radio button.
kControlUpButtonPart	20	Event occurred in the up (or left) scroll arrow of a scroll bar.
kControlDownButtonPart	21	Event occurred in the down (or right) button of a scroll bar.
kControlPageUpPart	22	Event occurred in the page-up part of a scroll bar.
kControlPageDownPart	23	Event occurred in the page-down part of a scroll bar.
kControlIndicatorPart	129	Event occurred in the scroll box of a scroll bar.

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, you should call `TrackControl` or `HandleControlClick` to follow and respond to the user's movements.

You can also use an action function to undertake additional actions as long as the user holds the mouse button down. Typically, action functions are used to continuously scroll the window's contents while the cursor is on a scroll arrow or gray area of a scroll bar, or in the scroll box of a live-feedback scroll bar. (As previously stated, you pass a pointer to this action function in the `actionProc` parameter of `TrackControl` or the `inAction` parameter of `HandleControlClick`).

`TrackControl` and `HandleControlClick` return the control's control part code if the user releases the mouse button while the cursor is still inside the control part, or `kControlNoPart` (0) if the cursor is outside the control part when the button is released. Your application should then respond appropriately.

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 the following functions to get and set control values. (Those appearing on a dark gray background were introduced with Mac OS 8.5.)

Function	Description
<code>GetControlValue</code>	Obtains the current value of a control.
<code>SetControlValue</code>	Changes the current value of a control.
<code>GetControlMinimum</code>	Obtains the minimum value of a control.
<code>SetControlMinimum</code>	Changes the minimum value of a control.

GetControlMaximum	Obtains the maximum value of a control.
SetControlMaximum	Changes the maximum value of a control.
GetControl32BitValue	Obtains the current value of a control.
SetControl32BitValue	Changes the current value of a control.
GetControl32BitMinimum	Obtains the minimum value of a control.
SetControl32BitMinimum	Changes the minimum value of a control.
GetControl32BitMaximum	Obtains the maximum value of a control.
SetControl32BitMaximum	Changes the maximum value of a control.

If your application uses a 32-bit control value, it should not use the pre-Mac OS 8.5 functions to obtain or change this value. In addition, your application should not attempt to obtain or change the value by accessing the `controlValue`, `controlMin`, or `controlMax` fields of the `ControlRecord` structure.

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` and `SetControlValue` 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 control. When your application creates a window, it should store handles for each scroll bar in an application-defined document structure 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 9.

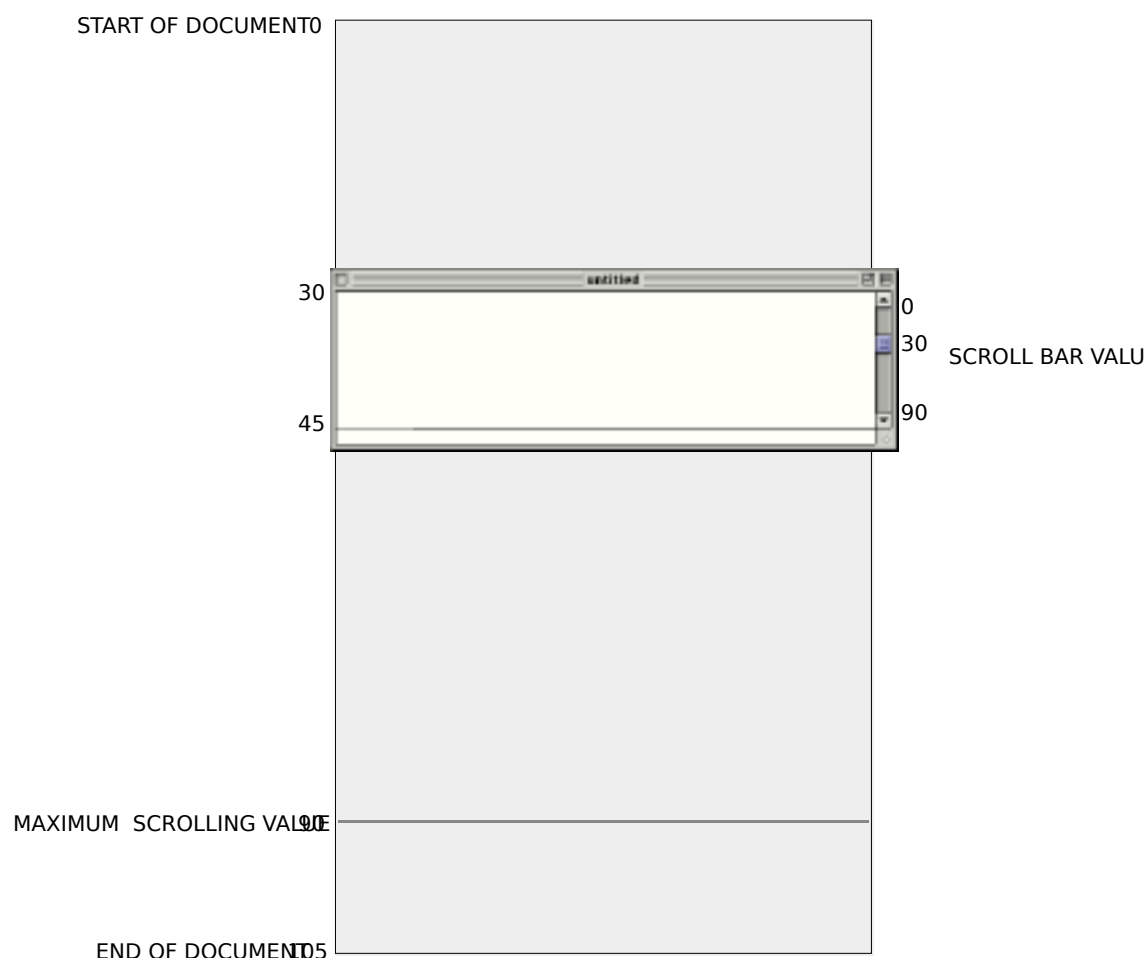


FIG 9 - SPATIAL RELATIONSHIP BETWEEN A DOCUMENT AND A WINDOW AND THEIR REPRESENTATION IN A SCROLL BAR

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.

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 function 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 10, 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).

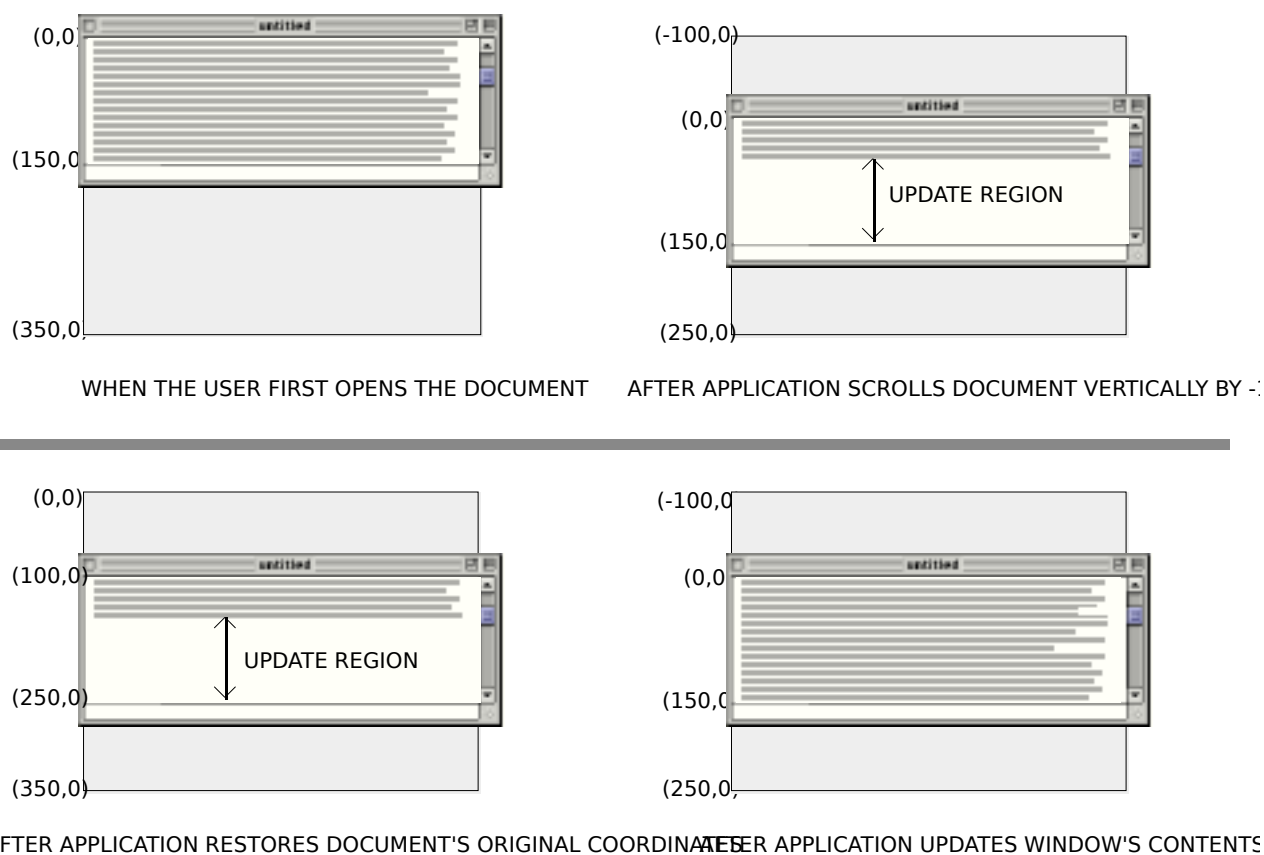


FIG 10 - SCROLLING A DOCUMENT IN A WINDOW

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 9, 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 10, 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 10.) 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 10.)

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

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 10.)

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 and Scrolling Using the List Manager

`TextEdit` is a collection of functions and data structures which you can use to provide your application with basic text editing capabilities. Chapter 19 — Text and `TextEdit` addresses, amongst other things, the scrolling of `TextEdit` documents.

For scrolling lists of graphic or textual information, your application can use the List Manager to implement scroll bars. (See Chapter 20 — Lists and Custom List Definition Functions.)

Main Control Manager Constants, Data Types and Functions Relevant to the Basic Controls, Primary Group Box (Text Title Variant) and User Panes

In the following:

- The constants, data types, and functions introduced with Mac OS 8 and the Appearance Manager are shown on a light gray background.
- The constants and functions introduced with Mac OS 8.5 are shown on a dark gray background.
- Those older constants, data types and functions affected by the introduction of Mac OS 8 and the Appearance Manager, but which may still be used in certain circumstances, are shown against a black background.

Constants

Control Definition IDs

kControlPushButtonProc	= 368
kControlPushButLeftIconProc	= 374
kControlPushButRightIconProc	= 375
kControlCheckBoxProc	= 369
kControlCheckBoxAutoToggleProc	= 371,
kControlRadioButtonProc	= 370
kControlRadioButtonAutoToggleProc	= 372
kControlScrollBarProc	= 384
kControlScrollBarLiveProc	= 386
kControlPopupButtonProc	= 400
kControlGroupBoxTextTitleProc	= 160
kControlUserPaneProc	= 256

Pop-up Menu Button Variation Codes

kControlPopupFixedWidthVariant	= 1
kControlPopupVariableWidthVariant	= 2
kControlPopupUseAddResMenuVariant	= 4
kControlPopupUseWFontVariant	= 8

Pop-up Title Characteristics

popupTitleBold	= \$0100;
popupTitleItalic	= \$0200;
popupTitleUnderline	= \$0400;
popupTitleOutline	= \$0800;
popupTitleShadow	= \$1000;
popupTitleCondense	= \$2000;
popupTitleExtend	= \$4000;
popupTitleNoStyle	= \$8000;

Control Variants

kControlNoVariant	= 0
kControlUsesOwningWindowsFontVariant	= 8

Control Part Codes

kControlNoPart	= 0
kControlLabelPart	= 1
kControlMenuPart	= 2
kControlButtonPart	= 10
kControlCheckBoxPart	= 11

kControlRadioButtonPart	= 11
kControlUpButtonPart	= 20
kControlDownButtonPart	= 21
kControlPageUpPart	= 22
kControlPageDownPart	= 23
kControlIndicatorPart	= 129
kControlDisabledPart	= 254
kControlInactivePart	= 255

Checkbox Value Constants

kControlCheckBoxUncheckedValue	= 0,
kControlCheckBoxCheckedValue	= 1,
kControlCheckBoxMixedValue	= 2

Radio Button Value Constants

kControlRadioButtonUncheckedValue	= 0,
kControlRadioButtonCheckedValue	= 1,
kControlRadioButtonMixedValue	= 2

Control Data Tag Constants

kControlPushButtonDefaultTag	= FOUR_CHAR_CODE('dflt')
kControlPushButtonCancelTag	= FOUR_CHAR_CODE('cncl')
kControlPopupButtonMenuHandleTag	= FOUR_CHAR_CODE('mhan')
kControlPopupButtonMenuIDTag	= FOUR_CHAR_CODE('mnid')
kControlPopupButtonExtraHeightTag	= FOUR_CHAR_CODE('exht')
kControlGroupBoxTitleRectTag	= FOUR_CHAR_CODE('trec')

Control Font Style Flag Constants

kControlUseFontMask	= 0x0001
kControlUseFaceMask	= 0x0002
kControlUseSizeMask	= 0x0004
kControlUseForeColorMask	= 0x0008
kControlUseBackColorMask	= 0x0010
kControlUseModeMask	= 0x0020
kControlUseJustMask	= 0x0040
kControlUseAllMask	= 0x00FF
kControlAddFontSizeMask	= 0x0100
kControlAddToMetaFontMask	= 0x0200

Meta Font Constants

kControlFontBigSystemFont	= -1
kControlFontSmallSystemFont	= -2
kControlFontSmallBoldSystemFont	= -3

Data Types

ControlPartCode : SInt16;

Control Structure

```
ControlRecord = PACKED RECORD
  nextControl : ControlHandle;      { Next Control. }
  contrlOwner : WindowPtr;          { Control's window. }
  contrlRect : Rect;                { Rectangle. }
  contrlVis : UInt8;                { 255 if visible, else 0. }
  contrlHilite : UInt8;             { Highlight state. }
  contrlValue : SInt16;             { Current setting. }
  contrlMin : SInt16;               { Minimum Setting. }
  contrlMax : SInt16;               { Maximum setting. }
  contrlDefProc : Handle;           { Control definition function. }
  contrldata : Handle;              { Data used by contrlDefProc. }
  contrlAction : ControlActionUPP; { Action function. }
  contrlRfCon : SInt32;             { Reference constant. }
  contrlTitle : Str255;             { Title. }
END;
```

```
ControlRecordPtr = ^ControlRecord;
ControlPtr = ^ControlRecord;
ControlHandle = ^ControlPtr;
```

Control Font Style Structure

```
ControlFontStyleRec = RECORD
```

```
  flags :      SInt16;
  font :      SInt16;
  size :      SInt16;
  style :     SInt16;
  mode :     SInt16;
  just :     SInt16;
  foreColor : RGBColor;
  backColor : RGBColor;
  END;
```

```
ControlFontStyleRecPtr = ^ControlFontStyleRec;
```

```
ControlFontStylePtr = ^ControlFontStyleRec;
```

Routines**Creating and Removing Controls**

```
FUNCTION NewControl(owningWindow: WindowPtr; {CONST}VAR boundsRect: Rect;
  controlTitle: ConstStr255Param; initiallyVisible: BOOLEAN; initialValue: SInt16;
  minimumValue: SInt16; maximumValue: SInt16; procID: SInt16;
  controlReference: SInt32): ControlHandle;
FUNCTION GetNewControl(resourceID: SInt16; owningWindow: WindowPtr): ControlHandle;
PROCEDURE DisposeControl(theControl: ControlHandle);
PROCEDURE KillControls(theWindow: WindowPtr);
```

Embedding Controls

```
OSErr      CreateRootControl(WindowPtr inWindow,ControlHandle * outControl);
OSErr      GetRootControl(WindowPtr inWindow,ControlHandle * outControl);
OSErr      EmbedControl(ControlHandle inControl,ControlHandle inContainer);
OSErr      AutoEmbedControl(ControlHandle inControl,WindowPtr inWindow);
OSErr      CountSubControl(ControlHandle inControl,SInt16 * outNumChildren);
OSErr      GetIndexedSubControl(ControlHandle inControl,SInt16 inIndex,
  ControlHandle * outSubControl);
OSErr      GetSuperControl(ControlHandle inControl,ControlHandle * outParent);
OSErr      SetControlSupervisor(ControlHandle inControl,ControlHandle inBoss);
OSErr      DumpControlHierarchy(WindowPtr inWindow,const FSSpec *inDumpFile);
```

Displaying and Manipulating Controls

```
PROCEDURE MoveControl(theControl: ControlHandle; h: SInt16; v: SInt16);
PROCEDURE SizeControl(theControl: ControlHandle; w: SInt16; h: SInt16);
PROCEDURE UpdateControls(theWindow: WindowPtr; updateRegion: RgnHandle);
PROCEDURE DrawControls(theWindow: WindowPtr);
PROCEDURE Draw1Control(theControl: ControlHandle);
PROCEDURE DrawControlInCurrentPort(inControl: ControlHandle);
FUNCTION IsControlActive(inControl: ControlHandle): BOOLEAN;
FUNCTION IsControlVisible(inControl: ControlHandle): BOOLEAN;
FUNCTION ActivateControl(inControl: ControlHandle): OSErr;
FUNCTION DeactivateControl(inControl: ControlHandle): OSErr;
PROCEDURE ShowControl(theControl: ControlHandle);
PROCEDURE HideControl(theControl: ControlHandle);
FUNCTION SetControlVisibility(inControl: ControlHandle; inIsVisible: BOOLEAN;
  inDoDraw: BOOLEAN): OSErr;
FUNCTION SendControlMessage(inControl: ControlHandle; inMessage: SInt16;
  inParam: SInt32): SInt32;
PROCEDURE DragControl(theControl: ControlHandle; startPoint: Point;
  {CONST}VAR limitRect: Rect; {CONST}VAR slopRect: Rect; axis: DragConstraint);
PROCEDURE HiliteControl(theControl: ControlHandle; hiliteState: ControlPartCode);
FUNCTION GetControlViewSize(theControl: ControlHandle): SInt32;
PROCEDURE SetControlViewSize(theControl: ControlHandle; newViewSize: SInt32);
```

Handling Events in Controls

```
FUNCTION FindControl(testPoint: Point; theWindow: WindowPtr;
  VAR theControl: ControlHandle): ControlPartCode;
FUNCTION FindControlUnderMouse(inWhere: Point; inWindow: WindowPtr;
  VAR outPart: SInt16): ControlHandle;
FUNCTION TrackControl(theControl: ControlHandle; startPoint: Point;
  actionProc: ControlActionUPP): ControlPartCode;
FUNCTION HandleControlClick(inControl: ControlHandle; inWhere: Point; inModifiers: SInt16;
  inAction: ControlActionUPP): SInt16;
```



```
FUNCTION TestControl(theControl: ControlHandle; testPoint: Point): ControlPartCode;
```

Accessing and Changing Control Settings and Data

```
FUNCTION GetControlValue(theControl: ControlHandle): SInt16;
PROCEDURE SetControlValue(theControl: ControlHandle; newValue: SInt16);
FUNCTION GetControlMinimum(theControl: ControlHandle): SInt16;
PROCEDURE SetControlMinimum(theControl: ControlHandle; newMinimum: SInt16);
FUNCTION GetControlMaximum(theControl: ControlHandle): SInt16;
PROCEDURE SetControlMaximum(theControl: ControlHandle; newMaximum: SInt16);
FUNCTION GetControl32BitValue(theControl: ControlHandle): SInt32;
PROCEDURE SetControl32BitValue(theControl: ControlHandle; newValue: SInt32);
FUNCTION GetControl32BitMaximum(theControl: ControlHandle): SInt32;
PROCEDURE SetControl32BitMaximum(theControl: ControlHandle; newMaximum: SInt32);
FUNCTION GetControl32BitMinimum(theControl: ControlHandle): SInt32;
PROCEDURE SetControl32BitMinimum(theControl: ControlHandle; newMinimum: SInt32);
PROCEDURE GetControlTitle(theControl: ControlHandle; VAR title: Str255);
PROCEDURE SetControlTitle(theControl: ControlHandle; title: ConstStr255Param);
FUNCTION GetControlReference(theControl: ControlHandle): SInt32;
PROCEDURE SetControlReference(theControl: ControlHandle; data: SInt32);
FUNCTION GetControlAction(theControl: ControlHandle): ControlActionUPP;
PROCEDURE SetControlAction(theControl: ControlHandle; actionProc: ControlActionUPP);
FUNCTION GetControlVariant(theControl: ControlHandle): ControlVariant;
FUNCTION SetControlData(inControl: ControlHandle; inPart: ControlPartCode;
    inTagName: ResType; inSize: Size; inData: Ptr): OSerr;
FUNCTION GetControlData(inControl: ControlHandle; inPart: ControlPartCode;
    inTagName: ResType; inBufferSize: Size; inBuffer: Ptr;
    VAR outActualSize: Size): OSerr;
FUNCTION GetControlDataSize(inControl: ControlHandle; inPart: ControlPartCode;
    inTagName: ResType; VAR outMaxSize: Size): OSerr;
FUNCTION SetControlVisibility(inControl: ControlHandle; inIsVisible: BOOLEAN;
    inDoDraw: BOOLEAN): OSerr;
```

Setting the Control Font Style

```
FUNCTION SetControlFontStyle(inControl: ControlHandle;
    {CONST}VAR inStyle: ControlFontStyleRec): OSerr;
```

Demonstration Program Controls1

```
{
// Controls1.p
//
// This program opens a kWindowFullZoomGrowDocumentProc window containing:
//
// • Four pop-up menu buttons (fixed width, variable width, add resource, and use window
//   font variants).
//
// • Three radio buttons auto-embedded in a primary group box (text title variant).
//
// • Three checkboxes auto-embedded in a primary group box (text title variant).
//
// • Four push buttons (two basic, one left colour icon variant, and one right colour
//   icon variant).
//
// • A vertical scroll bar (non live-feedback variant) and a horizontal scroll bar
//   (live-feedback variant).
//
// The window also contains a window header frame in which is displayed:
//
// • The menu items chosen from the pop-up menus.
//
// • The identity of a push button when that push button is clicked.
//
// • Scroll bar control values when the scroll arrows or gray areas of the scroll bars
//   are clicked and when the scroll box is dragged.
//
// The scroll bars are moved and resized when the user resizes or zooms the window;
// however, the scroll bars do not scroll the window content.
//
// A Demonstration menu allows the user to deactivate the group boxes in which the radio
// buttons and checkboxes are embedded.
//
// The program utilises the following resources:
//
```

Version 2.1

```
// • An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit, and Demonstration
//   menus, and the pop-up menus (preload, non-purgeable).
//
// • A 'WIND' resource (purgeable) (initially not visible).
//
// • 'CNTL' resources for the pop-up menus, group boxes, radio buttons, checkboxes,
//   buttons, and scroll bars (preload, purgeable) (initially visible).
//
// • Two 'cicn' resources (purgeable) for the colour icon variant buttons.
//
// • An 'hrct' resource and an 'hwin' resource (both purgeable), which provide help
//   balloons describing the various controls.
//
// • A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch,
//   and is32BitCompatible flags set.
//
// { } }

program Controls1;

//
.....
..... interfaces

uses

  { Universal Interfaces. }
  Appearance, Devices, Fonts, Menus, NumberFormatting, Processes, Resources, Sound,
  ToolUtils, LowMem;
//
.....
..... constants

const

rMenubar = 128;
rNewWindow = 128;
mApple = 128;
iAbout = 1;
mFile = 129;
iQuit = 11;
mDemonstration = 131;
iColour = 1;
iGrids = 2;

cPopupFixed = 128;
cPopupVariable = 129;
cPopupAddRes = 130;
cPopupWinFont = 131;
iCanberra = 1;
iLondon = 2;
iAuckland = 3;
iRome = 4;
cRadiobuttonRed = 132;
cRadiobuttonWhite = 133;
cRadiobuttonBlue = 134;
cCheckboxGrid = 135;
cCheckboxRulers = 136;
cCheckboxGridsnap = 137;
cGroupBoxColour = 138;
cGroupBoxGrids = 139;
cButton = 140;
cButtonDefault = 141;
cButtonLeftIcon = 142;
cButtonRightIcon = 143;
cScrollbarVert = 144;
cScrollbarHoriz = 145;

MAXLONG = $7FFFFFFF;

//
.....
..... types

type

DocRecord = record
  popupFixedHdl : ControlHandle;
  popupVariableHdl : ControlHandle;
  popupAddResHdl : ControlHandle;
```



```

end;

docRecordHdl^^.checkboxRulersHdl := GetNewControl(cCheckboxRulers, theWindowPtr);
if (docRecordHdl^^.checkboxRulersHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.checkboxGridSnapHdl := GetNewControl(cCheckboxGridsnap, theWindowPtr);
if (docRecordHdl^^.checkboxGridSnapHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.groupboxColourHdl := GetNewControl(cGroupBoxColour, theWindowPtr);
if (docRecordHdl^^.groupboxColourHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.groupboxGridsHdl := GetNewControl(cGroupBoxGrids, theWindowPtr);
if (docRecordHdl^^.groupboxGridsHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.buttonHdl := GetNewControl(cButton, theWindowPtr);
if (docRecordHdl^^.buttonHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.buttonDefaultHdl := GetNewControl(cButtonDefault, theWindowPtr);
if (docRecordHdl^^.buttonDefaultHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.buttonLeftIconHdl := GetNewControl(cButtonLeftIcon, theWindowPtr);
if (docRecordHdl^^.buttonLeftIconHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.buttonRightIconHdl := GetNewControl(cButtonRightIcon, theWindowPtr);
if (docRecordHdl^^.buttonRightIconHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.scrollbarVertHdl := GetNewControl(cScrollbarVert, theWindowPtr);
if (docRecordHdl^^.scrollbarVertHdl = nil) then
begin
ExitToShell;
end;

docRecordHdl^^.scrollbarHorizHdl := GetNewControl(cScrollbarHoriz, theWindowPtr);
if (docRecordHdl^^.scrollbarHorizHdl = nil) then
begin
ExitToShell;
end;

ignoredErr := AutoEmbedControl(docRecordHdl^^.radiobuttonRedHdl, theWindowPtr);
ignoredErr := AutoEmbedControl(docRecordHdl^^.radiobuttonWhiteHdl, theWindowPtr);
ignoredErr := AutoEmbedControl(docRecordHdl^^.radiobuttonBlueHdl, theWindowPtr);
ignoredErr := AutoEmbedControl(docRecordHdl^^.checkboxGridHdl, theWindowPtr);
ignoredErr := AutoEmbedControl(docRecordHdl^^.checkboxRulersHdl, theWindowPtr);
ignoredErr := AutoEmbedControl(docRecordHdl^^.checkboxGridSnapHdl, theWindowPtr);

ignoredErr := SetControlData(docRecordHdl^^.buttonDefaultHdl, kControlNoPart,
kControlPushButtonDefaultTag, sizeof(booleanData), @booleanData);

controlFontStyleStruc.flags := kControlUseFontMask;
controlFontStyleStruc.font := kControlFontSmallSystemFont;
ignoredErr := SetControlFontStyle(docRecordHdl^^.buttonLeftIconHdl,
controlFontStyleStruc);
controlFontStyleStruc.font := kControlFontSmallBoldSystemFont;
ignoredErr := SetControlFontStyle(docRecordHdl^^.buttonRightIconHdl,
controlFontStyleStruc);

```



```

inDrag: begin
  DragWindow(theWindowPtr, theEvent.where, qd.screenBits.bounds);
end;

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

inGrow: begin
  growRect := qd.screenBits.bounds;
  growRect.top := 336;
  growRect.left := 260;
  newSize := GrowWindow(theWindowPtr, theEvent.where, growRect);
  if (newSize <> 0) then
    begin
      SizeWindow(theWindowPtr, LoWord(newSize), HiWord(newSize), true);
      DoAdjustScrollBars(theWindowPtr);
      DoDrawMessage(theWindowPtr, true);
    end;
  end;

inZoomIn, inZoomOut: begin
  if (TrackBox(theWindowPtr, theEvent.where, partCode)) then
    begin
      SetPort(theWindowPtr);
      EraseRect(theWindowPtr^.portRect);
      ZoomWindow(theWindowPtr, partCode, false);
      DoAdjustScrollBars(theWindowPtr);
    end;
  end;

  otherwise begin
    end;
  end;
  { of case statement }
end;
{ of procedure DoMouseDown }

//  DoMenuChoice

procedure DoMenuChoice(menuChoice : SInt32);
  var
    menuID, menuItem : SInt16;
    itemName : Str255;
    daDriverRefNum : SInt16;
    menuHdl : MenuHandle;
    theWindowPtr : WindowPtr;
    docRecordHdl : DocRecordHandle;
    ignoredErr : OSErr;

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

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

  case menuID of

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

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

```



```

controlHdl : ControlHandle;
ignoredErr : OSerr;

begin
ignoredErr := GetRootControl(theWindowPtr, controlHdl);

if becomingActive then
begin
ignoredErr := ActivateControl(controlHdl);
DoDrawMessage(theWindowPtr, becomingActive);
end
else begin
ignoredErr := DeactivateControl(controlHdl);
DoDrawMessage(theWindowPtr, becomingActive);
end;
end;
{ of procedure DoActivateWindow }

// DoOSEvent DoOSEvent

procedure DoOSEvent({const} var theEvent : EventRecord);
begin
case BAnd(BSR(theEvent.message, 24), $000000FF) of

suspendResumeMessage: begin
glnBackground := BAnd(theEvent.message, resumeFlag) = 0;
DoActivateWindow(FrontWindow, not glnBackground);
end;

mouseMovedMessage: begin
end;

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

// DoInContent DoInContent

procedure DoInContent({const} var theEvent : EventRecord; theWindowPtr : WindowPtr);
var
docRecordHdl : DocRecordHandle;
controlHdl : ControlHandle;
controlValue, partCode : SInt16;
thePoint : Point;
ignoredCode : SInt16;

begin
docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

thePoint := Point(theEvent.where);
GlobalToLocal(thePoint);

partCode := FindControl(thePoint, theWindowPtr, controlHdl);
if (partCode <> 0) then
begin
if ((controlHdl = docRecordHdl^.popupFixedHdl) or
(controlHdl = docRecordHdl^.popupVariableHdl) or
(controlHdl = docRecordHdl^.popupAddResHdl) or
(controlHdl = docRecordHdl^.popupWinFontHdl)) then
begin
ignoredCode := TrackControl(controlHdl, thePoint, ControlActionUPP(-1));
controlValue := GetControlValue(controlHdl);
DoPopupMenuChoice(theWindowPtr, controlHdl, controlValue);
end
else if (controlHdl = docRecordHdl^.scrollbarVertHdl) then
begin
DoVertScrollbar(partCode, theWindowPtr, controlHdl, thePoint);
end
else if (controlHdl = docRecordHdl^.scrollbarHorizHdl) then
begin
ignoredCode := TrackControl(controlHdl, thePoint, gActionFunctionHorizUPP);
end
else begin
if (TrackControl(controlHdl, thePoint, nil) <> kControlNoPart) then
begin
if ((controlHdl = docRecordHdl^.radiobuttonRedHdl) or
(controlHdl = docRecordHdl^.radiobuttonWhiteHdl) or

```


Version 2.1

```
end;
end;
{ of procedure ActionFunctionVert }
```

```
// ActionFunctionHoriz
```

```
procedure ActionFunctionHoriz(controlHdl : ControlHandle; partCode : ControlPartCode);
var
  scrollDistance, controlValue : SInt16;
  valueString : Str255;
  theWindowPtr : WindowPtr;

begin
  gCurrentString := 'Horizontal Scroll Control Bar Value: ';

  if (partCode <> kControlIndicatorPart) then
    begin
      case partCode of

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

        kControlPageUpPart, kControlPageDownPart: begin
          scrollDistance := 55;
          end;

        otherwise begin
          end;
        end;
      { of case statement }

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

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

        DoMoveScrollBar(controlHdl, scrollDistance);
        end;

        NumToString(SInt32(GetControlValue(controlHdl)), valueString);
        gCurrentString := gCurrentString + valueString;
        theWindowPtr := controlHdl^^.contrlOwner;
        DoDrawMessage(theWindowPtr, true);
        end;
      { of procedure ActionFunctionHoriz }
```

```
// DoMoveScrollBar
```

```
procedure DoMoveScrollBar(controlHdl : ControlHandle; scrollDistance : SInt16);
var
  oldControlValue, controlValue, controlMax : SInt16;

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

  controlValue := oldControlValue - scrollDistance;

  if (controlValue < 0) then
    begin
      controlValue := 0;
    end
  else if (controlValue > controlMax) then
    begin
      controlValue := controlMax;
    end;

  SetControlValue(controlHdl, controlValue);
end;
{ of procedure DoMoveScrollBar }
```

```
// DoRadioButtons
```

```

procedure DoRadioButtons(controlHdl : ControlHandle; theWindowPtr : WindowPtr);
var
  docRecordHdl : DocRecordHandle;

begin
  docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

  SetControlValue(docRecordHdl^^.radiobuttonRedHdl, kControlRadioButtonUncheckedValue);
  SetControlValue(docRecordHdl^^.radiobuttonWhiteHdl, kControlRadioButtonUncheckedValue);
  SetControlValue(docRecordHdl^^.radiobuttonBlueHdl, kControlRadioButtonUncheckedValue);
  SetControlValue(controlHdl, kControlRadioButtonCheckedValue);
end;
  { of procedure DoRadioButtons }

// DoCheckboxes

procedure DoCheckboxes(controlHdl : ControlHandle);
begin
  if (GetControlValue(controlHdl) = 0) then
    begin
      SetControlValue(controlHdl, 1);
    end
  else begin
    SetControlValue(controlHdl, 0);
  end;
end;
  { of procedure DoCheckboxes }

// DoPushButtons

procedure DoPushButtons(controlHdl : ControlHandle; theWindowPtr : WindowPtr);
var
  docRecordHdl : DocRecordHandle;

begin
  docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

  if (controlHdl = docRecordHdl^^.buttonHdl) then
    begin
      gCurrentString := 'Button';
      DoDrawMessage(theWindowPtr, true);
    end
  else if (controlHdl = docRecordHdl^^.buttonDefaultHdl) then
    begin
      gCurrentString := 'Default Button';
      DoDrawMessage(theWindowPtr, true);
    end
  else if (controlHdl = docRecordHdl^^.buttonLeftIconHdl) then
    begin
      gCurrentString := 'Left Icon Button';
      DoDrawMessage(theWindowPtr, true);
    end
  else if (controlHdl = docRecordHdl^^.buttonRightIconHdl) then
    begin
      gCurrentString := 'Right Icon Button';
      DoDrawMessage(theWindowPtr, true);
    end;
end;
  { of procedure DoPushButtons }

// DoAdjustScrollBars

procedure DoAdjustScrollBars(theWindowPtr : WindowPtr);
var
  winRect : Rect;
  docRecordHdl : DocRecordHandle;

begin
  docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

  winRect := theWindowPtr^.portRect;

  HideControl(docRecordHdl^^.scrollbarVertHdl);
  HideControl(docRecordHdl^^.scrollbarHorizHdl);

  MoveControl(docRecordHdl^^.scrollbarVertHdl, winRect.right - 15, winRect.top + 25);
  MoveControl(docRecordHdl^^.scrollbarHorizHdl, winRect.left - 1, winRect.bottom - 15);

  SizeControl(docRecordHdl^^.scrollbarVertHdl, 16, winRect.bottom - 39);

```

Version 2.1

```
SizeControl(docRecordHdl^.scrollbarHorizHdl, winRect.right - 13, 16);

ShowControl(docRecordHdl^.scrollbarVertHdl);
ShowControl(docRecordHdl^.scrollbarHorizHdl);

SetControlMaximum(docRecordHdl^.scrollbarVertHdl,
    theWindowPtr^.portRect.bottom - theWindowPtr^.portRect.top - 25 - 15);
SetControlMaximum(docRecordHdl^.scrollbarHorizHdl,
    theWindowPtr^.portRect.right - theWindowPtr^.portRect.left - 15);
end;
{ of procedure DoAdjustScrollBars }

// DoDrawMessage
procedure DoDrawMessage(theWindowPtr : WindowPtr; inState : boolean);
var
    headerRect : Rect;
    windowHeight, stringWidth : SInt16;
    theState : SInt16;
    ignoredErr : OSErr;

begin
    theState := integer(inState);

    SetRect(headerRect, theWindowPtr^.portRect.left - 1, theWindowPtr^.portRect.top - 1,
        theWindowPtr^.portRect.right + 1, theWindowPtr^.portRect.top + 26);
    ignoredErr := DrawThemeWindowHeader(headerRect, theState);

    if (theState = kThemeStateActive) then
        begin
            ignoredErr := SetThemeTextColor(kThemeActiveWindowHeaderTextColor, gPixelDepth,
                glsColourDevice);
        end
    else
        begin
            ignoredErr := SetThemeTextColor(kThemeInactiveWindowHeaderTextColor, gPixelDepth,
                glsColourDevice);
        end;

    windowHeight := theWindowPtr^.portRect.right - theWindowPtr^.portRect.left;
    stringWidth := StringWidth(gCurrentString);
    MoveTo((windowHeight div 2) - (stringWidth div 2), 17);
    DrawString(gCurrentString);
end;
{ of procedure DoDrawMessage }

// DoPlaySound
procedure DoPlaySound(sndResourceName : Str255);
var
    soundHdl : SndListHandle;
    soundChanPtr : SndChannelPtr;
    ignoredErr : OSErr;


begin
    soundChanPtr := nil;

    soundHdl := SndListHandle(GetNamedResource('snd ', sndResourceName));
    if (soundHdl <> nil) then
        begin
            ignoredErr := SndPlay(soundChanPtr, soundHdl, true);
        end;
end;
{ of procedure DoPlaySound }

// DoGetDepthAndDevice
procedure DoGetDepthAndDevice;
var
    deviceHdl : GDHandle;

begin
    deviceHdl := LMGetMainDevice;
    gPixelDepth := deviceHdl^.gdPMap^.pixelSize;
    if (BitTst(Ptr(@deviceHdl^.gdFlags), gdDevType)) then
        begin
            glsColourDevice := true;
        end;
end;
{ of procedure DoGetDepthAndDevice }
```

```

//  main program block

begin
  //
  ..... initialise managers

  DoInitManagers;

  // .....
  create routine descriptors

  gActionFunctionVertUPP := NewControlActionProc(ControlActionProcPtr(@ActionFunctionVert));
  gActionFunctionHorizUPP := NewControlActionProc(ControlActionProcPtr(@ActionFunctionHoriz));

  // ..... cause the Appearance-compliant menu bar definition function to be called directly

  mainErr := RegisterAppearanceClient;

  // ..... set
  up menu bar and menus

  mainMenubarHdl := GetNewMBar(rMenubar);
  if (mainMenubarHdl = nil) then
    begin
      ExitToShell;
    end;
  SetMenuBar(mainMenubarHdl);
  DrawMenuBar;

  mainMenuHdl := GetMenuHandle(mApple);
  if (mainMenuHdl = nil) then
    begin
      ExitToShell;
    end
  else begin
    AppendResMenu(mainMenuHdl, 'DRVR');
  end;

  // ..... initial advisory text for window
  header

  gCurrentString := 'Balloon help is available';

  // ..... open a window, set font size, set Appearance-compliant colour/pattern for window

  mainWindowPtr := GetNewCWindow(rNewWindow, nil, WindowPtr(-1));
  if (mainWindowPtr = nil) then
    begin
      ExitToShell;
    end;

  SetPort(mainWindowPtr);
  TextSize(10);

  mainErr := SetThemeWindowBackground(mainWindowPtr, kThemeActiveDialogBackgroundBrush, true);

  // ..... get block for document structure, assign handle to window record refCon field

  mainDocRecordHandle := DocRecordHandle(NewHandle(sizeof(DocRecord)));
  if (mainDocRecordHandle = nil) then
    begin
      ExitToShell;
    end;

  SetWRefCon(mainWindowPtr, SInt32(mainDocRecordHandle));


  // ..... get controls, adjust scroll bars, and show window

  DoGetControls(mainWindowPtr);
  DoAdjustScrollBars(mainWindowPtr);
  ShowWindow(mainWindowPtr);

  // ..... get pixel depth and whether colour device for certain Appearance functions

  DoGetDepthAndDevice;

```

```
//  
..... enter eventLoop  
  
gDone := false;  
  
while not gDone do  
  begin  
    if WaitNextEvent(everyEvent, mainEvent, MAXLONG, nil) then  
      begin  
        DoEvents(mainEvent);  
      end;  
    end;  
  end.  
  { of main program block }  
  
// 
```

Demonstration Program Controls1 Comments

When this program is run, the user should:

- Choose Show Balloons from the Help menu and peruse the help balloons which are invoked when the mouse cursor is moved over the various controls.
- Choose items from each of the pop-up menu buttons, noting that the chosen item is displayed in the window header. (Also, in the case of the pop-up menu button titled Sound (which uses the add resource variant of the CDEF and adds the names of the system's 'snd ' resources to its menu), the chosen sound plays.)
- Click on the radio buttons, checkboxes, and push buttons, noting particularly that the radio button settings are mutually exclusive and that checkbox settings are not.
- Click in the scroll bar arrows and gray areas of the scroll bars, noting the control value changes displayed in the window header.
- Drag the scroll box of the vertical scroll bar (which uses the non-live-feedback CDEF variant), noting that only a ghosted outline is dragged and that the control value does not change until the mouse button is released.
- Drag the scroll box of the horizontal scroll bar (which uses the live-feedback CDEF variant), noting that the scroll box proper is dragged and that the control value is continually updated during the drag.
- Resize and zoom the window, noting (1) that the scroll bars are moved and resized in response to those actions and (2) the change in the maximum value of the scroll bars.
- Send the program to the background and bring it to the foreground, noting the changes to the appearance of the controls. (The program activates and deactivates the root control only; however, because all controls are embedded in the root control, all controls are activated and deactivated along with the root control.)
- Alternately hide and show the Colour primary group box by choosing the associated item in the Demonstration menu. (The program hides and shows the primary group box only; however, because the radio buttons are embedded in the primary group box, those controls are activated and deactivated along with the primary group box.)
- Alternately activate and deactivate the Grids primary group box by choosing the associated item in the Demonstration menu. (The program activates and deactivates the primary group box only; however, because the checkboxes are embedded in the primary group box, those controls are activated and deactivated along with the primary group box.) Also note the latency of the Show Rulers checkbox. It is deactivated at program launch, and retains that status when the primary group box is deactivated and then re-activated.

constants

The first block establishes constants representing a menu bar resource, a window resource, menu resources, menu IDs, and items for the drop-down menus. The second block establishes constants for control resources and items in the pop-up menu buttons (other than the add resource variant pop-up menu button). The penultimate line defines MAXLONG as the maximum possible long value. (This value will be assigned to WaitNextEvent's sleep parameter.)

type declarations

The data type DocRecord is a record comprising fields in which the handles to the control structures for the various controls will be stored. A handle to this record will be assigned to the window record's refCon field

Global Variables

gActionFunctionVertUPP and gActionFunctionHorizUPP will be assigned universal procedure pointers relating to action functions for the scroll bars.

`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.

`gPixelFormat` will be assigned the pixel depth of the main device. `gIsColourDevice` will be assigned true if the graphics device is a colour device and false if it is a monochrome device. The values in these two variables are required by the Appearance Manager function `SetThemeTextColor`.

DoGetControls

The routine `DoGetControls` creates the controls from the various 'CNTL' resources.

At the first line, the root control is created. The first control created must be always be the root control (which is implemented as a user pane).

A handle to the structure in which the handles to the control records will be stored is then retrieved. The following calls to `GetNewControl` create a control record for each control, insert the structure 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.

Because of the sequence in which the controls are created and initially drawn, the group boxes would ordinarily over-draw the radio buttons and checkboxes. However, the calls to `AutoEmbedControl` embed these latter controls in their respective group boxes, ensuring that they will be drawn after (or "on top of") the group boxes. (`AutoEmbedControl`, rather than `EmbedControl`, is used in this instance because the radio button rectangles are visually contained by their respective group box rectangles.)

The call to `SetControlData`, with `kControlPushButtonDefaultTag` passed in the third parameter causes the default outline to be drawn around the specified push button.

In the next block, the title fonts of the left colour icon variant and right colour icon variant push buttons are changed. Firstly, the flags and font fields of a control font style structure are assigned constants so that the following call to `SetControlFontStyle` will set the title font of the left colour icon variant push button to the small system font. The font field is then changed so that the second call to `SetControlFontStyle` will set the title font of the right colour icon variant push button to the small emphasized system font.

Lastly, the checkbox titled Rulers is disabled. This is for the purpose of the latency aspect of the demonstration.

DoEvents

`DoEvents` branches according to the event type reported. In the `keyDown/autoKey` case, the program will terminate if the Command-key equivalent for the Quit item in the File menu is pressed.

DoMouseDown

`DoMouseDown` branches according to the window part in which a `mouseDown` event occurs.

At the `inContent` case, if the window in which the mouse-down occurred is the front window, and since all of the controls are located in the window's content region, a call to the application-defined routine `DoInContent` is made.

The `inGrow` case is of particular significance to the scroll bars. `GrowWindow` 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, `SizeWindow` is called to draw the window in its new size, and an application-defined routine is called to erase, move, resize, and redraw the scroll bars and reset the control's maximum value according to the new size of the window. (The following line is incidental to the demonstration. It simply redraws the window header frame and text in the window.)

The `inZoomIn/InZoomOut` case is also of significance to the scroll bars. If the call to `TrackBox` returns a non-zero value, the content region is erased, `ZoomWindow` is called to redraw the window in its new state, and an application-defined routine is called to hide, move, resize, and redraw the scroll bars.

DoMenuChoice

`DoMenuChoice` handles user choices from the drop-down menus.

The `mDemonstration` case handle the Demonstration menu. Firstly, handles to that menu and to the window's "document" record are obtained.

If the menu item is the Colour item, `IsControlVisible` is called to determine the current visibility status of the Colour group box. If it is visible, the call to `HideControl` hides the group box and its embedded radio buttons; also, the menu item is changed to "Show Colour". If it is not visible, `ShowControl` is called and the menu item is changed to Hide Colour.

At the else if, if the menu item is the Grids item, the same general sequence takes place in respect of the Grids group box. This time, however, `IsControlActive` is used to determine whether the control is active or inactive, and `ActivateControl` and `DeactivateControl` are called, and the menu item toggled, as appropriate. Note that, because of latency, the application does not have to "remember" that one of the embedded checkboxes was deactivated at program start. The Control Manager does the remembering.

DoUpdate

DoUpdate is called whenever the application receives an update event for its window. Between the usual calls to BeginUpdate and EndUpdate, the window's graphics port is set as the current port for drawing, and UpdateControls is called to draw those controls intersecting the current visible region (which, between the BeginUpdate and EndUpdate calls, equates to the update region). The line preceding the UpdateControls call is incidental to the demonstration. It simply redraws the window header frame and text in the window.

DoActivate

DoActivate is called whenever the application receives an activate event for its window. 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 routine DoActivateWindow.

DoActivateWindow

DoActivateWindow branches 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 routine DoOSEvent.)

At the first line, GetRootControl gets a handle to the window's root control.

If the window is becoming active, ActivateControl is called to activate the root control. Since all other controls are embedded in the root control, all controls will be activated by this call.

If the window is about to become inactive, DeactivateControl is called to deactivate the root control. Since all other controls are embedded in the root control, all controls will be deactivated by this call.

The calls to DoDrawMessage are incidental to the demonstration. They simply redraw the window header frame and text in the window in the appropriate mode (inactive or active).

DoOSEvent

DoOSEvent handles operating system events. If the event is a suspend or resume event, a variable is then set to indicate whether the program is coming to the foreground or is about to be sent to the background. This variable is passed in the call to DoActivateWindow. (Recall that the doesActivateOnFGSwitch flag is set in the 'SIZE' resource.)

DoInContent

DoInContent further processes mouse-down events in the content region. Since the content region of the window contains nothing but controls, this function is really just the main dispatching point for the further handling of those controls.

The first line gets the handle to the "document" structure containing the handles to the various control structures. The second line converts the mouse coordinates in the event structure's where field from global coordinates to the local coordinates required in the following call to FindControl. (FindControl is used here rather than the new routine FindControlUnderMouse because there is no requirement to get a handle to a control even if no part was hit and no requirement to determine whether a mouse-down event has occurred in a deactivated control.)

If there is a control at the cursor location at which the mouse button is released, the control handle returned by the FindControl call is first compared with the handles to the pop-up menu controls stored in the window's "document" structure. If a match is found, TrackControl is called with ControlActionUPP(-1) passed in the actionProc parameter so as to cause an action function within the control's CDEF to be repeatedly invoked while the mouse button remains down. When TrackControl returns, the control value is obtained by a call to GetControlValue and an application-defined routine is called to perform further handling

Note that TrackControl, rather than the new function HandleControlClick, is used in this program because none of the controls require modifier keys to be passed in. (Of course, HandleControlClick would work just as well (with 0 passed in the inModifiers parameter).)

If the control handle returned by FindControl does not match the pop-up controls' handles, it is then tested against the handles to the vertical and horizontal scroll bar control structures. If it matches the handle to the vertical scroll bar (which uses the non-live-feedback CDEF variant), the application-defined routine DoVertScrollbar is called to perform further handling. If it matches the handle to the horizontal scroll bar (which uses the live-feedback CDEF variant), TrackControl is called with a Universal Procedure Pointer (UPP) passed in the actionProc parameter. The effect of this is that the UPP will be assigned to the cntrlAction field of the control structure and thus the application-defined action function to which the UPP relates will be repeatedly called while the mouse button remains down.

ROUTINE DESCRIPTORS AND UNIVERSAL PROCEDURE POINTERS

This call to TrackControl, incidentally, is your first encounter with source code which differs from that which would have applied prior to the introduction of the Power Macintosh and its PowerPC microprocessor.

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

```
FUNCTION TrackControl(theControl: ControlHandle; startPoint: Point;
                    actionProc: ProcPtr): ControlPartCode;
```

Notice that the actionProc parameter is of type ProcPtr (procedure pointer). The third parameter is thus a pointer to a routine, specifically, an action function. That being the case, the subject call to TrackControl would have looked like this in the days when source code was intended only for compilation for 680x0 microprocessors:

```
TrackControl(theControl,theEvent.where, @gActionFunctionHoriz);
```

With the introduction of the Universal Interfaces, the declaration for TrackControl changed to this:

```
FUNCTION TrackControl(theControl: ControlHandle; startPoint: Point;
                    actionProc: ControlActionUPP): ControlPartCode;
```

Notice that the actionProc parameter is now of type ControlActionUPP. (UPP stands for Universal Procedure Pointer).

As a result of this change:

- For source code intended for compilation as either PowerPC code or 680x0 code, a call to create each required routine descriptor needs to be made as shown in the main function of this program and the the pointer (UPP) to that routine descriptor needs to be passed in the actionProc parameter of the subject TrackControl call.
- For source code intended only for compilation as 680x0 code, the routine descriptor does not need to be created and only a simple procedure pointer (the address of the action function) may be passed in the actionProc parameter, as in the example call above.

If the handle returned by FindControl does not match the handles to any of the pop-up menu buttons or scroll bars, it must be a handle to one of the other controls. In this case, TrackControl is called, with the procPtr field set to that required for push buttons, radio buttons, and checkboxes (that is, nil). If the cursor is still within the control when the mouse button is released, the handle is compared to, in sequence, the handles to the radio buttons, the checkboxes, and the push buttons. If a match is found, the appropriate application-defined routine is called to perform further handling.

DoPopupMenuChoice

DoPopupMenuChoice further handles mouse-downs in the pop-up menu buttons. The code reflects the fact that the Sound pop-up menu button, which uses the add resource variant of the CDEF, and whose 'CNTL' resource causes the names of the system's 'snd' resources to be appended to its associated menu, needs to be handled differently from the others.

If the control handle passed to this function matches the handle to the Sound pop-up menu button's control structure, the handle to the control's associated 'MENU' structure is retrieved and used, together with the control's value (that is, the menu item), in the call to GetMenuItemText. This call returns, in the last parameter, the menu item text. This string is then passed to an application-defined routine which plays the chosen 'snd' resource. The call to DoDrawMessage draws the menu item text in the window header frame.

Note that the menu handle is not obtained in the same way as would apply in the case of a drop-down menu (that is, calling GetMenuHandle with the menu ID). In this case, GetControlData is called with the tag constant kControlPopupMenuHandleTag passed in the second parameter, the menu handle being returned in the fifth parameter.

This is a good example of why the new data access mechanism was introduced with Mac OS 8 and the Appearance Manager. In the past, the only way to allow access to control-specific information was to create a handle to hold such data, place it in the `controlData` field of the control structure, and publish the interface. For example, pop-up menu buttons place a handle to a block of private information (called the pop-up menu private data structure) in the `controlData` field. This structure contains two fields: a handle to the pop-up menu button's menu structure (`mHandle`), and the menu ID (`mID`). In the past, before the new data access mechanism was introduced, the menu handle would have to have been retrieved like this:

```
popupPrivateDataHdl : PopupPrivateDataHandle;

popupPrivateDataHdl := PopupPrivateDataHandle(controlHdl^^.controlData);
menuHdl := popupPrivateDataHdl^^.mHandle;
```

The new data access mechanism means that controls can now allow the outside world to access their specialised data without exposing how it is stored.

If the control handle passed to this function does not match that for the Sound pop-up menu button, the routine branches on the control value. In this case, the further handling that would normally take place here is replaced in this demonstration by the simple drawing of some text (which is identical with the menu item's text) in the window header frame.

DoVertScrollbar

`DoVertScrollbar` is called from `DoInContent` in the case of a mouse-down in the vertical scroll bar (which uses the non-live-feedback variant of the CDEF).

The copying of text to the global variable `gCurrentString` is incidental to the demonstration.

At the next line, the routine branches on the control part code. If the control part code was the scroll box (that is, the indicator), `TrackControl` is called with the `procPtr` parameter set to that required for the scroll box of non-live-feedback scroll bars (that is, `NULL`). If the user did not move the cursor outside the control before releasing it, the `if` block executes, retrieving the new control value, converting it to a string, appending that string to the string currently in `gCurrentString`, and drawing `gCurrentString` in the window header. (In a real application, calculation of the distance and direction to scroll, and the scrolling itself, would take place inside this `if` block.)

If the mouse down was in the gray area or one of the scroll arrows, `TrackControl` is called with a Universal Procedure Pointer (UPP) passed in the `actionProc` parameter. The effect of this is that the application-defined action function to which the UPP relates will be repeatedly called while the mouse button remains down.

At this point, another brief digression is necessary.

ACTION FUNCTIONS

Action functions (sometimes called *hook functions* or *call-back functions*) refer to the ability of a Toolbox function to call an application-defined routine during its execution, thus extending the features of the function.

The following is for information only, and can be safely ignored by Pascal programmers:

Toolbox calls use Pascal calling conventions, and C and Pascal are different in their conventions on the 680x0 processor. An action function can be written in C; however, in order to account for the difference in calling conventions, it must be declared using the *pascal* keyword. (Note the first line of the function `actionFunctionVert`.)

ActionFunctionVert

`ActionFunctionVert` is the action function called by `TrackControl` at the bottom of `DoVertScrollbar`. Because it is repeatedly called by `TrackControl` while the mouse button remains down, the scrolling such a function would perform in a real application continues repeatedly until the mouse button is released (provided the cursor remains within the scroll arrow or gray area).

The copying of text to the global variable `gCurrentString` is incidental to the demonstration.

The "if `controlPartCode` then" line ensures that, if the cursor is not still inside the scroll arrow or gray area, the action function exits and all scrolling ceases until the user brings the cursor back within the scroll arrow or gray area, causing a non-zero control part code to be again received. The following occurs only when the cursor is within the control.

The function branches on the control part code. If the mouse-down is in a scroll arrow, the variable `scrollDistance` is set to 2. If it is in a gray area, `scrollDistance` is set to 55. (In this simple demonstration, these are just arbitrary values. In a real application, you would assign an appropriate value in the case of the scroll arrows, and assign a calculated value (based primarily on current window height) in the case of the gray areas.)

The next block convert the value in `scrollDistance` to the required negative value if the user is scrolling down rather than up.

The next block defeats any further scrolling action if, firstly, the down scroll arrow is being used and the "document" is at the maximum scrolled position or, secondly, the up scroll arrow is being used and the "document" is at the minimum scrolled position.

The distance to scroll having been set, the call to the application-defined routine DoMoveScrollBar moves the scroll box the appropriate distance in the appropriate direction and update the control's value accordingly. This means, of course, that the scroll box is being continually moved, and the control's value continually updated, while the mouse button remains down.

In this demonstration, the remaining action is to retrieve the current value of the control, convert it to a string, append it to the string currently in gCurrentString, and draw gCurrentString in the window header frame. (In a real application, the actual scrolling of the window's contents would be effected here.)

ActionFunctionHoriz

ActionFunctionHoriz is the action function passed in the actionProc parameter of the TrackControl call in DoInContent arising from a mouse-down in the horizontal scroll bar. This action function differs from that for the vertical scroll bar because the horizontal scroll bar uses the live-feedback variant of the CDEF.

The principal differences are that action functions for live-feedback scroll bars must continually scroll the window's contents, not only while the mouse button remains down in the scroll arrows and gray areas, but also while the scroll box is being dragged. Accordingly, this function, unlike the action function for the vertical scroll bar, is also called while the mouse button remains down in the scroll box.

The copying of text to the global variable gCurrentString is incidental to the demonstration.

If the mouse-down occurred in the scroll box, the code which sets up the scroll distance, adjusts the sign of the scroll distance according to whether the scroll is left or right, prevents scrolling beyond the minimum and maximum scroll values, and calls DoMoveScrollBar to move the scroll box and update the control's value, is bypassed. The call to DoMoveScrollBar is bypassed because, in live-feedback, the CDEF moves the scroll box and updates the control's value when the mouse-down is in the scroll box.

In this demonstration, the action taken after the main block of code has been bypassed (mouse-down in the scroll box) or executed (mouse-down in the scroll arrows or gray area) is to retrieve the current value of the control, convert it to a string, append it to the string currently in gCurrentString, and draw gCurrentString in the window header frame. (In a real application, the actual scrolling of the window's contents would be effected here.)

DoMoveScrollBar

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

The first two lines retrieve the control's current value and maximum value. The next line calculates the new control value by subtracting the distance to scroll received from the calling action function from the current control value. The next four lines prevent the control's value from being set lower or higher than the control's minimum and maximum values respectively. The call to SetControlValue sets the new control value and repositions the scroll box.

DoRadioButtons

DoRadioButtons is called when the mouse-down is within a radio button. The first three calls to SetControlValue set all radio buttons to the off state. The final call sets the radio button under the mouse to the on state.

DoCheckboxes

DoCheckboxes is called when the mouse-down is within a checkbox. The single line simply toggles the current value of the control.

DoPushButtons

DoPushButtons is called when the mouse-down is within a push button. In this demonstration, the only action taken is to draw the identity of the push button in the window header frame.

DoAdjustScrollBars

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

At the first line, a handle to the window's "document" structure is retrieved from the window structure's refCon field.

At the next line, 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 cause a very slight flickering of the scroll bars. To prevent this, the scroll bars will be hidden while these two functions are executing.

The calls to HideControl hide the scroll bars. The calls to MoveControl erase the scroll bars, offset the contrlRect fields of their control structures, and redraw the scroll bars within the offset rectangle. SizeControl hides the scroll bars (in this program they are already hidden), adjusts the contrlRect fields of their control structures, and redraws the scroll bars within their new rectangles. The calls to ShowControl then show the scroll bars.

In this demonstration, the remaining lines set the new maximum values for the scroll bars according to the new height and width of the window. No attempt is made to calculate the required new control value to ensure that the (non-existent) document remains in the same scrolled position after the zoom or resize. In a real application, this, plus the calculation of the maximum value according to, for example, the line height of text content as well as the new window height, are matters that would need to be attended to in this function.

DoDrawMessage, DoPlaySound, and DoGetDepthAndDevice

DoDrawMessage, DoPlaySound, and DoGetDepthAndDevice are incidental to the demonstration.

main program block

After the system software managers are initialised, two calls to NewControlActionProc create a routine descriptor for each of two action functions associated with the scroll bars. (If this program was required to be compiled as 68K code only, these routine descriptors would not be required.)

The next block sets up the drop-down menus.

The string copied to gCurrentString, which will be changed at various points in the code, will be drawn in the window header frame.

The next block opens a window, makes the window's graphics port the current port, and sets the size of the window font to 10. This latter is because one of the pop-up menus will use the window font and Geneva 10pt is the ideal size for small pop-up menu text. The call to SetThemeWindowBackground sets an Appearance-compliant background colour/pattern for the window. The window's background will be similar to that applying to dialog boxes, which is appropriate for a window containing nothing but controls.

The call to NewHandle gets a relocatable block the size of one DocRecord structure. The handle to the block is assigned to the window record's refCon field by the call to SetWRefCon.

In the next block, DoGetControls creates and draws the controls, DoAdjustScrollBars resizes and locates the scroll bars, and sets their maximum value, according to the dimensions of the window's port rectangle, and ShowWindow makes the window visible.

The call to the application-defined routine DoGetDepthAndDevice determines the current pixel depth of the graphics port, and whether the current graphics device is a colour device, and assigns the results to the global variables gPixelDepth and glsColourDevice.

The main event loop is then entered, and continues until gDone is set to true.

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 Controls2

```
{
// Controls2.p
//
// This program:
//
// • Opens a kWindowDocumentProc window with a two horizontal scroll bars, each of
//   which relates to the picture displayed immediately above it.
//
// • Allows the user to horizontally scroll the pictures within the window using the
//   scroll box, the scroll arrows and the gray area of each scroll bar.
//
// The top scroll bar uses the non-live-feedback variant of the scroll bar CDEF. The
// bottom scroll bar uses the live-feedback variant.
//
// If the target is the PowerPC target, and if Mac OS 8.5 or later is present, the scroll
// bar scroll boxes are made proportional.
//
// In this program, the action functions are set using the function SetControlAction.
// (In Controls1, the action functions were passed in TrackControl's actionProc
// parameter.)
//
// The program utilises the following resources:
```

```

//
// • An 'MBAR' resource, and 'MENU' resources for Apple, File and Edit (preload, non-
//   purgeable).
//
// • A 'WIND' resource (purgeable) (initially visible).
//
// • Two 'CNTL' resource for the horizontal scroll bars (purgeable).
//
// • A 'PICT' resource containing the picture to be scrolled (non-purgeable).
//
// • A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch,
//   and is32BitCompatible flags set.
//
// ◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇ }
program Controls2;

//
.....
..... interfaces

uses

  { Universal Interfaces. }
  Appearance, Devices, Fonts, GestaltEq, Processes, Sound, ToolUtils;

//
.....
..... constants

const

rMenuBar = 128;
rNewWindow = 128;
rPictureNonLive = 128;
rPictureLive = 129;
mApple = 128;
iAbout = 1;
mFile = 129;
iQuit = 11;
cScrollbarNonLive = 128;
cScrollbarLive = 129;

MAXLONG = $7FFFFFFF;

//
.....
..... type declarations

type

DocRecord = record
  scrollbarNonLiveHdl : ControlHandle;
  scrollbarLiveHdl : ControlHandle;
end;

DocRecordPtr = ^DocRecord;
DocRecordHandle = ^^DocRecord;

//
.....
..... global variables

var

gActionFuncNonLiveUPP : ControlActionUPP;
gActionFuncLiveUPP : ControlActionUPP;
gDone : Boolean;
gInBackground : Boolean;
gPictRectNonLive, gPictRectLive : Rect;
gPictHandleNonLive, gPictHandleLive : PicHandle;

// ..... main
program block variables

mainMenuBarHdl : Handle;
mainMenuHdl : MenuHandle;
mainWindowPtr : WindowPtr;
mainDocRecordHdl : DocRecordHandle;
osError : OSErr;

```

Version 2.1

```
response, viewSize: SInt32;  
mainEvent : EventRecord;
```

```
//
```

```
..... routine declarations
```

```
procedure DoInitManagers; forward;  
procedure DoEvents({const} var theEvent : EventRecord); forward;  
procedure DoMouseDown({const} var theEvent : EventRecord); forward;  
procedure DoUpdate({const} var theEvent : EventRecord); forward;  
procedure DoActivate({const} var theEvent : EventRecord); forward;  
procedure DoActivateWindow(theWindowPtr : WindowPtr; becomingActive : Boolean); forward;  
procedure DoOSEvent({const} var theEvent : EventRecord); forward;  
procedure DoMenuChoice(menuChoice : SInt32); forward;  
procedure DoInContent({const} var theEvent : EventRecord; theWindowPtr : WindowPtr); forward;  
procedure DoNonLiveScrollBars(partCode : ControlPartCode; theWindowPtr : WindowPtr;  
    controlHdl : ControlHandle; mouseXY : Point); forward;  
procedure DoMoveScrollBar(controlHdl : ControlHandle; scrollDistance : SInt16); forward;  
  
procedure ActionFuncNonLive(controlHdl : ControlHandle; partCode : ControlPartCode); forward;  
procedure ActionFuncLive(controlHdl : ControlHandle; partCode : ControlPartCode); forward;
```

```
//   
DoInitManagers
```

```
procedure DoInitManagers;  
    var  
        osError : OSErr;  
  
    begin  
        MaxApplZone;  
        MoreMasters;  
  
        InitGraf(@qd.thePort);  
        InitFonts;  
        InitWindows;  
        InitMenus;  
        TEInit;  
        InitDialogs(nil);  
  
        InitCursor;  
        FlushEvents(everyEvent, 0);  
  
        osError := RegisterAppearanceClient;  
  
    end;  
    { of procedure DoInitManagers }
```

```
//   
DoEvents
```

```
procedure DoEvents({const} var theEvent : EventRecord);  
    begin  
        case (theEvent.what) of  
  
            mouseDown: begin  
                DoMouseDown(theEvent);  
            end;  
  
            updateEvt: begin  
                DoUpdate(theEvent);  
            end;  
  
            activateEvt: begin  
                DoActivate(theEvent);  
            end;  
  
            osEvt: begin  
                DoOSEvent(theEvent);  
                HiliteMenu(0);  
            end;  
  
            otherwise begin  
                end;  
            end;  
        { of case statement }  
    end;  
    { of procedure DoEvents }
```

```
//   
DoMouseDown
```



```

procedure DoMouseDown({const} var theEvent : EventRecord);
var
  theWindowPtr : WindowPtr;
  partCode : SInt16;

begin
  partCode := FindWindow(theEvent.where, theWindowPtr);

  case partCode of

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

    inContent: begin
      if (theWindowPtr <> FrontWindow) then
        begin
          SelectWindow(theWindowPtr);
        end
      else begin
        DoInContent(theEvent, theWindowPtr);
      end;
    end;

    inDrag: begin
      DragWindow(theWindowPtr, theEvent.where, qd.screenBits.bounds);
    end;

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

    otherwise begin
      end;
  end;
  { of case statement }
end;
{ of procedure DoMouseDown }

//  DoUpdate

procedure DoUpdate({const} var theEvent : EventRecord);
var
  theWindowPtr : WindowPtr;
  docRecordHdl : DocRecordHandle;

begin
  theWindowPtr := WindowPtr(theEvent.message);
  docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

  BeginUpdate(theWindowPtr);

  SetPort(theWindowPtr);

  UpdateControls(theWindowPtr, theWindowPtr^.visRgn);

  SetOrigin(GetControlValue(docRecordHdl^.scrollbarNonLiveHdl), 0);
  DrawPicture(gPictHandleNonLive, gPictRectNonLive);
  SetOrigin(0, 0);

  SetOrigin(GetControlValue(docRecordHdl^.scrollbarLiveHdl), 0);
  DrawPicture(gPictHandleLive, gPictRectLive);
  SetOrigin(0, 0);

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

//  DoActivate

procedure DoActivate({const} var theEvent : EventRecord);
var
  theWindowPtr : WindowPtr;
  becomingActive : Boolean;

begin
  theWindowPtr := WindowPtr(theEvent.message);

```



```

        begin
        gDone := true;
        end;
    end;

    otherwise begin
    end;
end;
{ of case statement }

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

// DoInContent
procedure DoInContent(const theEvent : EventRecord; theWindowPtr : WindowPtr);
var
    docRecordHdl : DocRecordHandle;
    partCode : ControlPartCode;
    controlHdl : ControlHandle;
    thePoint : Point;
    ignoredCode : SInt16;

begin
    docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

    thePoint := theEvent.where;
    GlobalToLocal(thePoint);

    partCode := FindControl(thePoint, theWindowPtr, controlHdl);
    if (partCode <> kControlNoPart) then
    begin
        if (controlHdl = docRecordHdl^^.scrollbarNonLiveHdl) then
        begin
            DoNonLiveScrollBars(partCode, theWindowPtr, controlHdl, thePoint);
        end
        else if (controlHdl = docRecordHdl^^.scrollbarLiveHdl) then
        begin
            ignoredCode := TrackControl(controlHdl, thePoint, ControlActionUPP(-1));
        end;
    end;
end;
{ of procedure DoInContent }

// DoNonLiveScrollBars
procedure DoNonLiveScrollBars(partCode : ControlPartCode; theWindowPtr : WindowPtr;
    controlHdl : ControlHandle; mouseXY : Point);
var
    docRecordHdl : DocRecordHandle;
    oldControlValue : SInt16;
    scrollDistance : SInt16;
    updateRgnHdl : RgnHandle;
    ignoredCode : SInt16;

begin
    docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

    case partCode of
        kControlIndicatorPart: begin
            oldControlValue := GetControlValue(controlHdl);
            if (TrackControl(controlHdl, mouseXY, nil) <> kControlNoPart) then
            begin
                scrollDistance := oldControlValue - GetControlValue(controlHdl);
                if (scrollDistance <> 0) then
                begin
                    if (controlHdl = docRecordHdl^^.scrollbarNonLiveHdl) then
                    begin
                        updateRgnHdl := NewRgn;
                        ScrollRect(gPictRectNonLive, scrollDistance, 0, updateRgnHdl);
                        InvalRgn(updateRgnHdl);
                        DisposeRgn(updateRgnHdl);
                    end;
                end;
            end;
        end;
    end;

    kControlUpButtonPart, kControlDownButtonPart, kControlPageUpPart,

```

Version 2.1

```
kControlPageDownPart: begin
  if (controlHdl = docRecordHdl^^.scrollbarNonLiveHdl) then
    begin
      ignoredCode := TrackControl(controlHdl, mouseXY, ControlActionUPP(-1));
    end;
  end;

  otherwise begin
    end;
  end;
  { of case statement }
end;
{ of procedure DoNonLiveScrollBars }
```

```
// ActionFuncNonLive
```

```
procedure ActionFuncNonLive(controlHdl : ControlHandle; partCode : ControlPartCode);
var
  theWindowPtr : WindowPtr;
  docRecordHdl : DocRecordHandle;
  scrollDistance : SInt16;
  controlValue : SInt16;
  updateRgnHdl : RgnHandle;

begin
  if (partCode <> kControlNoPart) then
    begin
      theWindowPtr := controlHdl^^.ctrlOwner;
      docRecordHdl := DocRecordHandle(GetWRefCon(theWindowPtr));

      case partCode of

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

        kControlPageUpPart, kControlPageDownPart: begin
          scrollDistance := (theWindowPtr^.portRect.right - theWindowPtr^.portRect.left) - 10;
        end;

        otherwise begin
          end;
        end;
        { of case statement }

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

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

      DoMoveScrollBar(controlHdl, scrollDistance);

      if ((partCode = kControlUpButtonPart) or
        (partCode = kControlDownButtonPart)) then
        begin
          updateRgnHdl := NewRgn;
          ScrollRect(gPictRectNonLive, scrollDistance, 0, updateRgnHdl);
          InvalRgn(updateRgnHdl);
          DisposeRgn(updateRgnHdl);
          BeginUpdate(theWindowPtr);
        end;

      SetOrigin(GetControlValue(docRecordHdl^^.scrollbarNonLiveHdl), 0);
      DrawPicture(gPictHandleNonLive, gPictRectNonLive);
      SetOrigin(0, 0);

      if ((partCode = kControlUpButtonPart) or
        (partCode = kControlDownButtonPart)) then
        begin
          EndUpdate(theWindowPtr);
        end;
      end;
    end;
  end;
```



```

gPictRectLive := gPictHandleLive^^.picFrame;

OffsetRect(gPictRectLive, 0, 201);


// ..... if Mac OS 8.5 or later present, set up for proportional scroll boxes
{$ifc TARGET_CPU_PPC}
  osError := Gestalt(gestaltSystemVersion, response);

  if ((osError = noErr) and (response >= $00000850)) then
    begin
      viewSize := gPictHandleNonLive^^.picFrame.right -
                  gPictHandleNonLive^^.picFrame.left;
      SetControlViewSize(mainDocRecordHdl^^.scrollbarNonLiveHdl,200);
      viewSize := gPictHandleLive^^.picFrame.right -
                  gPictHandleLive^^.picFrame.left;
      SetControlViewSize(mainDocRecordHdl^^.scrollbarLiveHdl,200);
    end;
{$endc}

//
..... enter eventLoop

gDone := false;

while not gDone do
  begin
    if (WaitNextEvent(everyEvent, mainEvent, MAXLONG, nil)) then
      begin
        DoEvents(mainEvent);
      end;
  end;
end.
{ of main program block }

// 

```

Demonstration Program Controls2 Comments

This program is basically an extension of the scroll bars aspects of the demonstration program Controls1 in that, unlike the scroll bars in Controls1, the scroll bars in this program actually scroll the contents of the window.

When the program is run, the user should scroll the pictures by dragging the scroll boxes, clicking in the scroll bar gray areas, clicking in the scroll arrows and holding the mouse button down while the cursor is in the gray areas and scroll arrows. The user should note, when scrolling with the scroll boxes, that the top scroll bar uses the non-live-feedback variant of the scroll bar CDEF and the bottom scroll bar uses the live-feedback variant, this latter to facilitate the program's live-scrolling of the bottom picture.

The pictures scrolled in this demonstration are 600 pixels wide and 185 pixels high, the window "pane" for each picture and scroll bar is 200 pixels wide by 200 pixels high, the 'CNTL' resources sets the control maximum values to 400, and the control rectangles specified in the 'CNTL' resource locate the scroll bars in the correct position in the non-resizable, non-zoomable window.

As an incidental aspect of the demonstration, two different methods are used to scroll the pictures when the scroll arrows are being used. In the top picture, at each pass through the action function, the pixels are scrolled using ScrollRect, the "vacated" area is invalidated, and only this vacated area is redrawn. In the bottom picture, at each pass through the action function, the whole visible part of the picture is redrawn. The user should note that the first method results in some flickering in the "vacated" area when the picture is scrolled, and that the second method eliminates this flickering at the cost of some horizontal "tearing" of the picture. (This latter should be recalled at Chapter 23 — Miscellany, where the subject of VBL tasks is addressed. See the end of the Demonstration Program Comments section.) Note that either method may be used for either scroll bar.

The following comments are limited to those areas which are significantly different from the same areas in the demonstration program Controls1

DoUpdate

In the two blocks which draw the pictures, the first call to SetOrigin sets the window origin to the current scroll position, that is, to the position represented by the control's current value, thus ensuring that the correct part of the picture will be drawn by the call to DrawPicture. The second call to SetOrigin resets the window's origin to (0,0).

DoInContent

DoInContent establishes whether a mouse-down event was in one of the scroll bars and, if so, branches accordingly.

The call to GlobalToLocal converts the global coordinates of the mouse-down, stored in the where field of the event structure, to the local coordinates required by FindControl. If the call to FindControl returns a non-zero result, the mouse-down was in a scroll bar.

As in Controls1:

- If the mouse-down was in the non-live-feedback scroll bar, an application-defined routine is called to further handle the mouse-down event.
- If the mouse-down was in the live-feedback scroll bar, TrackControl is called with ControlActionUPP(-1) passed in the actionProc parameter. This means that the application-defined routine associated with the UPP assigned by SetControlAction to the contrlAction field of the control structure will be continually called while the mouse button remains down.

DoNonLiveScrollBars

DoNonLiveScrollBars is similar to its sister function in Controls1 except that it actually scrolls the window's contents.

At the first line, the function branches on the control part code:

- If the mouse-down was in the scroll box (that is, the indicator), the control's value at the time of the mouse-down is retrieved. Control is then handed over to TrackControl, 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. If the user moved the scroll box, the picture's pixels are scrolled by the specified scroll distance in the appropriate direction, and the "vacated" area of the window following the scroll is added to the (currently empty) window update region. This means that an update event will be generated for the window and that the re-draw of the picture will be attended to in the DoUpdate function.
- If the mouse-down was in a scroll arrow or gray area, more specifically in one of the non-live-feedback's scroll bar's scroll arrows or gray areas, TrackControl takes control until the user releases the mouse button. The third parameter in the TrackControl call means that the application-defined routine associated with the UPP assigned by SetControlAction to the contrlAction field of the control structure will be continually called while the mouse button remains down.

ActionFunctionNonLive

ActionFunctionNonLive is the action function for the non-live-feedback scroll bar. 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.

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

A pointer to the window record for the window which "owns" this control is retrieved from the control record's `cntrlOwner` field, and the handle to the record structure is retrieved from that window record's `refCon` field.

If the control part being used by the user to perform the scrolling is one of the scroll arrows, the distance to scroll (in pixels) is set to 2. 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. (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.)

The first block following the switch converts the distance to scroll to the required negative value if the user is scrolling towards the right. The second block defeats 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.

With the scroll distance determined, the call to the application-defined routine `DoMoveScrollBar` adds/subtracts the distance to scroll to/from the control's current value and repositions the scroll box accordingly.

At this stage, the picture scrolling takes place. If scrolling is being effected using the scroll arrows, `ScrollRect` scrolls 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 then added to the window's update region (previously empty) by the call to `InvalRgn`, and `BeginUpdate` is called to ensure that (1) only the "vacated" area will be redrawn and (2) the update region is cleared.

Regardless of whether the picture is being scrolled using the scroll arrows or the gray areas, `SetOrigin` is then called to reset 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).

Finally, if `BeginUpdate` was called prior to the draw (that is, scrolling is being effected using the scroll arrows), `EndUpdate` is called.

ActionFunctionLive

ActionFunctionLive is the action function for the live-feedback scroll bar.

The principal differences between this action function and the previous one are that action functions for live-feedback scroll bars must continually scroll the window's contents, not only while the mouse button remains down in the scroll arrows and gray areas, but also while the scroll box is being dragged. Accordingly, this action function, unlike the action function for the non-live-feedback scroll bar, is also called while the mouse button remains down in the scroll box.

If the mouse-down occurred in the scroll box, the code which sets up the scroll distance, adjusts the sign of the scroll distance according to whether the scroll is left or right, prevents scrolling beyond the minimum and maximum scroll values, and calls `DoMoveScrollBar` to move the scroll box and update the control's value, is bypassed. The call to `DoMoveScrollBar` is bypassed because, the live-feedback variant of the CDEF moves the scroll box and updates the control's value when the mouse-down is in the scroll box.

After the if block has been bypassed (mouse-down in the scroll box) or executed (mouse-down in the scroll arrows or gray area), the window contents are "scrolled". Regardless of whether the picture is being scrolled using the scroll box, the scroll arrows, or the gray areas, `SetOrigin` is called to reset the window origin so that that part of the picture represented by the current scroll position is drawn by the call to `DrawPicture`. After the correct part of the picture is drawn, the window origin is reset to (0,0).

Note that this alternative approach to re-drawing the picture when scrolling is being effected using the scroll arrows, has not been dictated by the fact that this is a live-feedback action function. Either of these two approaches will work in both live-feedback and non-live-feedback action functions.

DoMoveScrollBar

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

main program block

After the block for the "document" record is created, two calls to `GetNewControl` allocate memory for the control records, insert the controls into the window's control list and draw the controls.

Following each call to `GetNewControl`, `SetControlAction` is called with a Universal Procedure Pointer (UPP) passed in the `actionProc` parameter. The effect of this is that the UPP will be assigned to the `cntrlAction` field of the control structure and thus the application-defined action function to which the UPP relates will be repeatedly called while the mouse button remains down. As a consequence of using `SetControlAction`, `ControlActionUPP(-1)` will be passed in `TrackControl's`

Version 2.1

actionProc parameter. (This approach differs from the alternative method used in Controls1, where the UPP is passed in TrackControl's actionProc parameter.)

Note that no root control is created in this program; accordingly, the two controls will be activated and deactivated individually.

In the next block, two 'PICT' resources are loaded, the associated handles being assigned to two global variables. In each case, the picture structure's picFrame field (a Rect) is copied to a global variable. In the case of the second picture, this rectangle is then offset downwards by 201 pixels. (Note that the two 'PICT' resources were created so that the top and left fields of the picFrame Rect are both zero.)

The next block is applicable only to the PowerPC target, and the if block executes only if Mac OS 8.5 or later is present. The width of each picture in pixels is determined and passed in the newViewSize parameter of calls to SetControlViewSize. (This value is in the same units of measurement as are used for the scroll bar minimum, maximum, and current values.) This makes the scroll boxes proportional provided that the user has selected Smart Scrolling on in the Option tab of the Appearance control panel.