7

INTRODUCTION TO CONTROLS

Demonstration Programs: Controls1 and Controls2

Introduction

On Mac OS 8/9, 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. All but one or two of the remaining controls are addressed at Chapter 14. The list box control is addressed at Chapter 22.

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 dialogs and alerts. In this latter case, the Dialog Manager works with the Control Manager behind the scenes. The creation and handling of controls in dialogs and alerts will be addressed at Chapter 8 and in the demonstration program associated with Chapter 14. The creation and handling areas of this chapter are limited to the use of the Control Manager to create and handle controls in document or utility 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 window's window object.

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.

Available Standard Controls

The complete range of available standard controls is as follows.

Control	Description	Variants
Push button	A control that appears on the screen as a rounded rectangle with a title centred inside. When the user clicks a push button, the application performs the action described by the button's title. Examples include completing operations defined by a dialog and acknowledging an error message in an alert.	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** (Mac OS 8/9) or **scroller** (Mac OS X). The rest of the scroll bar is called the **gray area** (Mac OS 8/9) or **track** (Mac OS X). 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 (Mac OS 8/9 only). With normal bevel. With large bevel (Mac OS 8/9 only). 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 right or left and down. When the disclosure triangle points to the right or left, 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 bar.	A control indicating that a lengthy operation is occurring. Two types of progress bar can be used: an **indeterminate** progress bar reveals that an operation is occurring but does not indicate its duration; a **determinate** progress bar displays how much of the operation has been completed. Progress bars are also used as relevance bars on Mac OS 8/9.	(One variant only. However, the progress bar can be made determinate or non-determinate via a call to SetControlData.)
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.)
Chasing arrows	A control that 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.	Large, north facing. Small, north facing. Large, south facing. Small, south facing. Large, east facing. Small, east facing. Large, west facing. Small, west facing.
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 that may or may not contain a title. It is used to provide a well-defined area in a dialog 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 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 surrounded by a rectangular frame.	(One variant only.)
Pop-up arrow	A control that simply draws the pop-up glyph.	Large, east-facing. Large, west-facing. Large, north-facing. Large, south-facing. Small, east -facing. Small, west -facing. Small, north -facing. Small, south -facing.
Placard	A rectangular control used to delineate an area in which information may be displayed.	(One variant only.)
Clock	A control that combines the features of little arrows and an edit text field into a control which displays a date and/or time.	Displays hours, minutes. Displays hours, minutes, 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.	Normal For passwords. For 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 rectangular control that is positioned along the top of a window's content region and which is used to delineate an area in which information may be displayed.	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.
Data Browser	A control that implements a user interface component for browsing (optionally) hiearchical data structures. **Note:** This control is not addressed in this book.	
Disclosure button	A button used to hide or show specific content. Available on Mac OS X only.	(One variant only)
Edit Unicode text	Similar to the edit text controls, except that it handles Unicode text. Available on Mac OS X only.	(One variant only)
Relevance bar.	A control that indicates a level of relevance. Available on Mac OS X only.	(One variant only.)
Round button	Similar to a push button, except that it is round. Available on Mac OS X only.	Normal size. Large size.

Definition of the Term "Controls"

On Mac OS 8/9, prior to the introduction of Mac OS 8 and the Appearance Manager, a control was defined as an "on-screen object 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 Mac OS 8/9 theme-compliance introduced with the Appearance Manager. For example, using the provided separator line "control" to draw separator lines will ensure that those lines are drawn with the correct Platinum appearance (and, on Mac OS X, the correct Aqua "look") in both the activated and deactivated modes.

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 feature of some of the controls introduced with Mac OS 8 and the Appearance Manager.)

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 that 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 Figs 1 and 2.

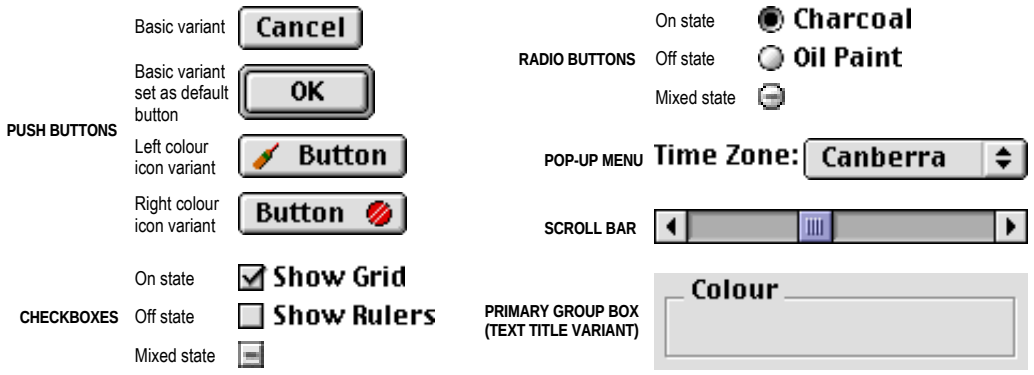

FIG 1 - THE BASIC CONTROLS AND THE PRIMARY GROUP BOX (TEXT TITLE VARIANT) (MAC OS 8/9)

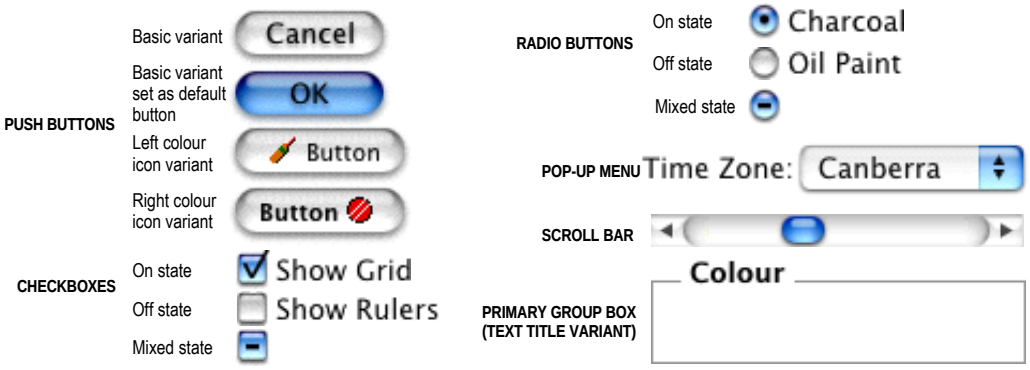

FIG 2 - THE BASIC CONTROLS AND THE PRIMARY GROUP BOX (TEXT TITLE VARIANT) (MAC OS X)

The Control Definition Function

Control definition functions (CDEFs), which are stored as resources of type `'CDEF'`, determine the appearance and behaviour of a control.

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.

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

Note that the a single 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 alerts and dialogs.

In windows and dialogs containing 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. On Mac OS 8/9, the default push button is outlined. On Mac OS X it is coloured and pulsing. (See Fig 1.)

In your application, pressing the Enter and Return keys should result in the same action as clicking on the the default push button.

Checkboxes

Checkboxes are typically used in dialogs. Checkboxes provide alternative choices and act like toggle switches. Each checkbox must have a title, which should reflect two clearly opposite states.

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 that 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.

Auto-Toggling Variant

When the auto-toggling variant is being used, checkboxes 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 checkbox's new state. There is no need to programmatically change the control's value after tracking successfully.

Radio Buttons

Like checkboxes, radio buttons are typically used in dialogs. Unlike checkboxes, radio buttons within a group are mutually exclusive.

Radio buttons are typically grouped. Each group must have a label indicating the kind of choices offered by the group, and each button must have a title indicating what it does.

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 that 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.

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.

Scroll Bars

Scroll bars scroll a document within a window. Scroll bars have **scroll arrows** at each end and a movable **scroll box** (Mac OS 8/9) or **scroller** (Mac OS X). The part of a scroll bar not occupied by the scroll arrows and scroll box/scroller is called the **gray area** (Mac OS 8/9) or **track** (Mac OS X). When the user clicks in a scroll arrow or gray area/track, or drags the scroll box/scroller, your application must scroll the document as appropriate.

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/scroller, the scroll box/scroller moves and the document scrolls as the user moves the mouse. (Without live scrolling, only a ghosted image of the scroll box/scroller moves. In addition, the document is only scrolled, and the scroll box/scroller proper is only redrawn at its new location, when the user releases the mouse button.)

Scroll Arrows and Gray Area/Track

When the scroll arrows are clicked, your application should move the scroll box/scroller 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 specified direction. (In a text document, one unit would typically be one line of text.)

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

Scroll Box/Scroller

Live Feedback Variant Not Used

When the live feedback variant is not being used, and when the user drags the scroll box/scroller, the Control Manager redraws the scroll box/scroller 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 scroll box/scroller 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/scroller, the Control Manager continually redraws the scroll box/scroller, and continually returns the control's position (that is, its **value**) as the scroll box/scroller moves. Once again, your application uses this value to display the appropriate portion of the document.

Proportional Scroll Boxes/Scrollers

A proportional scroll box/scroller 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.

Those of your application's scroll boxes/scrollers created from `'CNTL'` resources will appear as proportional scroll boxes/scrollers 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/scroller once your application supplies this information. (On Mac OS 8/9, the user must also have selected **Smart**

Scrolling on in the **Options** tab in the Appearance control panel.) For scroll bars created programmatically using the function `CreateScrollBarControl`, you pass the size of the view area in the `viewSize` parameter.

The following functions are relevant to proportional scroll boxes/scrollers:

Function	*Description*
`GetControlViewSize`	Obtains the size of the content to which a control's size is proportioned.
`SetControlViewSize`	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 an alternative method of providing the user with a list of choices. The items in a pop-up menu button's menu should be mutually exclusive. Pop-up menus should be used to provide a choice of attributes, and should not be used to provide additional commands.

Primary Group Boxes (Text Title Variant)

A primary group box (text title variant) is a control that consists of a rectangular 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 (callback) functions, known as user pane functions, which perform actions such as drawing, hit testing, etc.

Activating, Deactivating, Hiding, Showing, Enabling, and Disabling Controls

Activating and Deactivating Controls

A control can be either **active** or **inactive**. A control should be made inactive when it is inappropriate for your application to respond to a mouse-down event in that control. The Control Manager displays inactive controls in a dimmed state. The Control Manager displays inactive basic controls and inactive primary group boxes as shown at Figs 3 and 4.

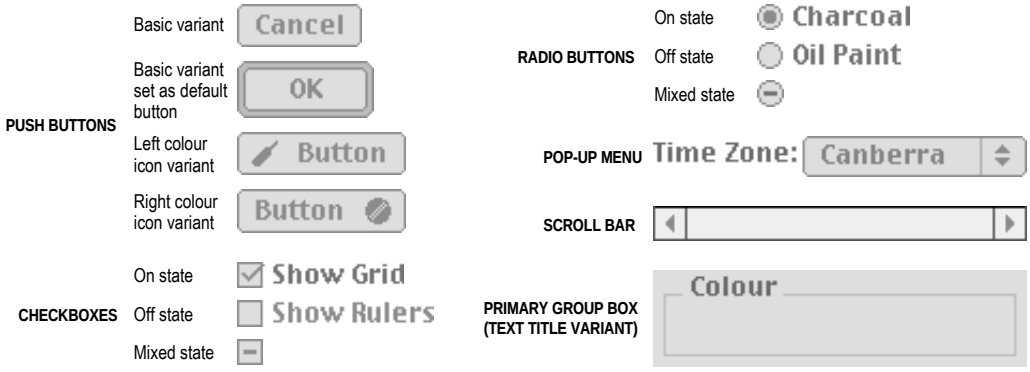

FIG 3 - THE BASIC CONTROLS AND THE PRIMARY GROUP BOX (TEXT TITLE VARIANT) IN INACTIVE MODE (MAC OS 8/9)

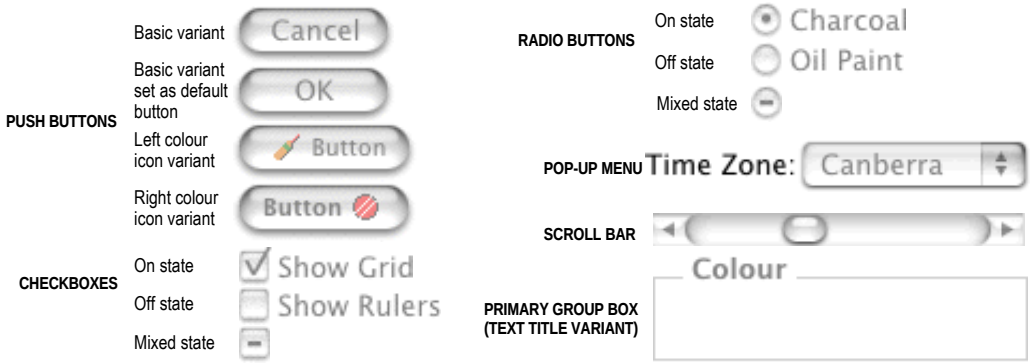

FIG 4 - THE BASIC CONTROLS AND THE PRIMARY GROUP BOX (TEXT TITLE VARIANT) IN INACTIVE MODE (MAC OS X)

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. (But see also Enabling and Disabling Controls, below.)
- 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`.

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`.

Enabling and Disabling Controls

`EnableControl` and `DisableControl` may be used to enable and disable controls. `IsControlEnabled` may be used to determine whether a control is enabled or disabled.

A distinction needs to be made between an disabled control and a deactivated control. A deactivated control is simply a control in a deactivated window. (All controls in a window should be deactivated when the window is deactivated.) A disabled control, on the other hand, is a control which the user should not currently be able to manipulate. The activation state of the window is immaterial in this case.

A "delete" button, for example, should be disabled when nothing is currently selected. As another example, controls in floating windows should never be deactivated, simply because floating windows

themselves should never be deactivated. However, a control in a floating window should be disabled if the user should not currently be able to manipulate it.

It follows that a control in a deactivated window must be deactivated and may be either disabled or enabled.

A problem here is that, as of the time of writing, `EnableControl`, `DisableControl`, and `IsControlEnabled` are only available on Mac OS X. Thus, on Mac OS 8/9, the only way to deny user access to a control in an active window is to deactivate it.

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/scroller (live feedback variant of the scroll bar CDEF being used) or moving a ghosted image of the scroll box/scroller (live feedback variant not being used) when the user drags it.

`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 newer 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

As previously stated, controls (or, more usually, a group of controls) may be embedded in **embedder controls**.

The Root Control

The embedding of controls in a window requires that the window have a **root control**. 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**.

On Mac OS X, a root control is created automatically in all document windows that have at least one other control. On Mac OS 8/9, however, you must explicitly create the root control in document windows 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 5. By acting on the group box alone, you can then activate, deactivate, hide, show, and move all four controls as a group.

`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 5, `AutoEmbedControl` would be more appropriate.

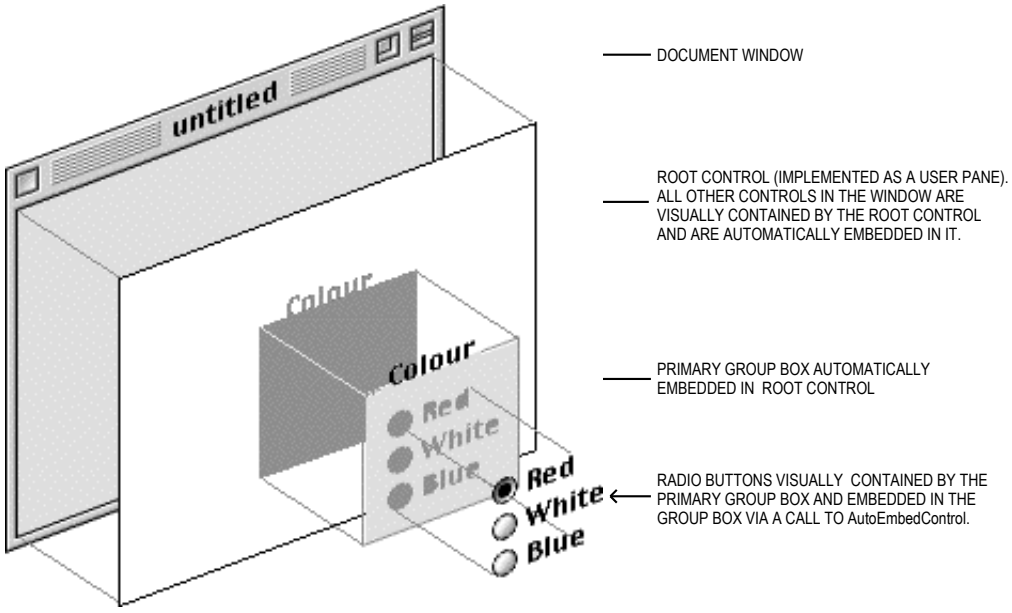

FIG 5 - THE ROOT CONTROL AND EMBEDDED CONTROLS

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 it they were created.

In the example at Fig 5, 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 5 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 activated, 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

Getting and setting control data is essentially 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 reference or a structure.

Control data tag constants are passed in the third parameter of the getter and setter functions `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*
`kControlPushButtonDefaultTag`	Causes a push button to be drawn with the appearance of the default push button, or returns whether the push button is drawn with the default push button appearance. Data type returned or set: `Boolean`
`kControlPushButtonCancelTag`	Gets or sets whether a push button plays the Cancel button theme sound instead of the normal push button theme sound. Data type returned or set: `Boolean`. Default is `false`.
`kControlPopupButtonMenuRefTag`	Gets or sets the menu reference for a pop-up menu. Data type returned or set: `MenuRef`
`kControlPopupButtonMenuIDTag`	Gets or sets the menu ID for a pop-up menu button. Data type returned or set: `SInt16`
`kControlPopupButtonExtraHeightTag`	Gets or sets the amount of extra white space in a pop-up menu button. Data type returned or set: `SInt16`. Default is `0`.
`kControlGroupBoxTitleRectTag`	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`
`kControlScrollBarShowsArrowsTag`	Mac OS X only. Specifies whether the scroll arrows are to be drawn or not. Data type set: `Boolean`

The Control Object

The Control Manager stores information about individual controls in opaque data structures called **control objects**. The data type `ControlRef` is defined as a pointer to a control object:

```
typedef struct OpaqueControlRef*  ControlRef;
```

Accessor Functions

Accessor functions are provided to access the information in control objects. The accessor functions are as follows:

Function	*Description*
`GetControlOwner`	Gets a reference to the window in which the specified control is located.
`SetControlOwner`	Assigns the specified control to a specified window.
`GetControlBounds`	Sets the specified control's enclosing rectangle.
`SetControlBounds`	Gets the specified control's enclosing rectangle.
`IsControlVisible`	Determines if the specified control is currently hidden or showing.
`SetControlVisibility`	Shows or hides the specified control.
`IsControlHilited`	Determines if the specified control is currently highlighted.
`HiliteControl`	Changes active/inactive status of a control or highlights a specified part of a control.
`GetControlHilite`	Determines whether the specified control is currently highlighted.
`GetControlValue`	Gets the specified control's value (16 bit) as set by `SetControlValue`.
`SetControlValue`	Sets the specified control's value (16 bit).

GetControl32BitValue	Sets the specified control's value as set by SetControl32BitValue.
SetControl32BitValue	Gets the specified control's value.
GetControlMaximum	Gets the specified control's maximum value (16 bit) as set by SetControlMaximum.
SetControlMaximum	Sets the specified control's maximum value (16 bit).
GetControl32BitMaximum	Gets the specified control's maximum value as set by SetControl32BitMaximum.
SetControl32BitMaximum	Sets the specified control's maximum value.
GetControlMinimum	Gets the specified control's minimum value (16 bit) as set by SetControlMinimum.
SetControlMinimum	Sets the specified control's minimum value (16 bit).
GetControl32BitMinimum	Gets the specified control's minimum value as set by SetControl32BitMinimum.
SetControl32BitMinimum	Sets the specified control's minimum value.
GetControlDataHandle	Gets a handle to data specific to a particular control type.
SetControlDataHandle	Sets a handle to data specific to a particular control type.
GetControlAction	Gets the universal procedure pointer to the specified control's action function.
SetControlAction	Sets a universal procedure pointer to the specified control's action function.
GetControlReference	Gets the specified control's reference constant.
SetControlReference	Sets the specified control's reference constant.
GetControlTitle	Gets the specified control's title.
SetControlTitle	Set the specified control's title.
GetControlDefinition	Gets the specified control's definition function.
GetControlPopupMenuHandle	Gets the specified popup menu button control's menu reference.
SetControlPopupMenuHandle	Sets the menu reference for a popup menu button control.
GetControlPopupMenuID	Gets the specified pop-up menu button control's menu ID.
SetControlPopupMenuID	Sets the menu ID for a pop-up menu button control.

Creating Controls

When you use the Dialog Manager to implement push buttons, radio buttons, checkboxes or pop-up menu buttons in alerts or dialogs, Dialog Manager functions automatically use Control Manager functions to create the controls for you.

For document and utility windows, you can create controls from 'CNTL' resources or you can create them programmatically.

Creating Controls From 'CNTL' Resources

You create controls from 'CNTL' resources using GetNewControl. GetNewControl, which takes a 'CNTL' resource ID and a reference to the window object, creates a control object from the information in the resource, adds the control object to the control list for your window, and returns a reference to the control.

Before you can create a control using GetNewControl, you must, of course, first create the necessary 'CNTL' resource. 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 6 shows the structure of a compiled 'CNTL' resource and how it "feeds" the control object.

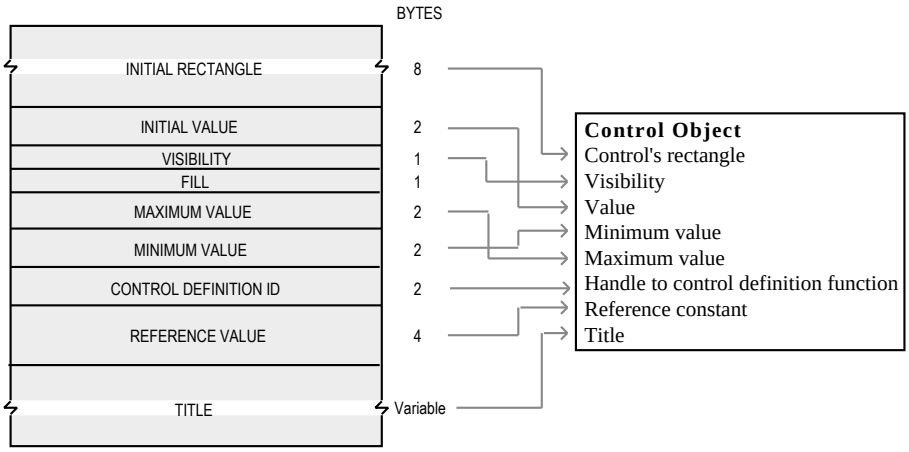

FIG 6 - STRUCTURE OF A COMPILED CONTROL ('CNTL') RESOURCE

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

Field	*Description*
INITIAL RECTANGLE	A rectangle that determines the control's size and location. It is specified in local coordinates.
INITIAL VALUE	Initial value for the control. (See Values for Controls, below).
VISIBILITY	Visibility of the control. When this field contains the value `true`, `GetNewControl` draws the control immediately. When this field contains `false`, the application must use `ShowControl` when the time has come to display the control.
MAXIMUM VALUE	Maximum value of the control. (See Values for Controls, below).
MINIMUM VALUE	Minimum value for the control. (See Values for Controls, below).
CONTROL DEFINITION ID	The control definition ID. (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. The value must be between –32768 and 32767.	Whatever maximum value is appropriate. The value must be between –32768 and 32767. 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. These fields are thus said to be "overloaded".

Creating 'CNTL' Resources Using Resorcerer

Fig 7 shows a `'CNTL'` resource being created with Resorcerer.

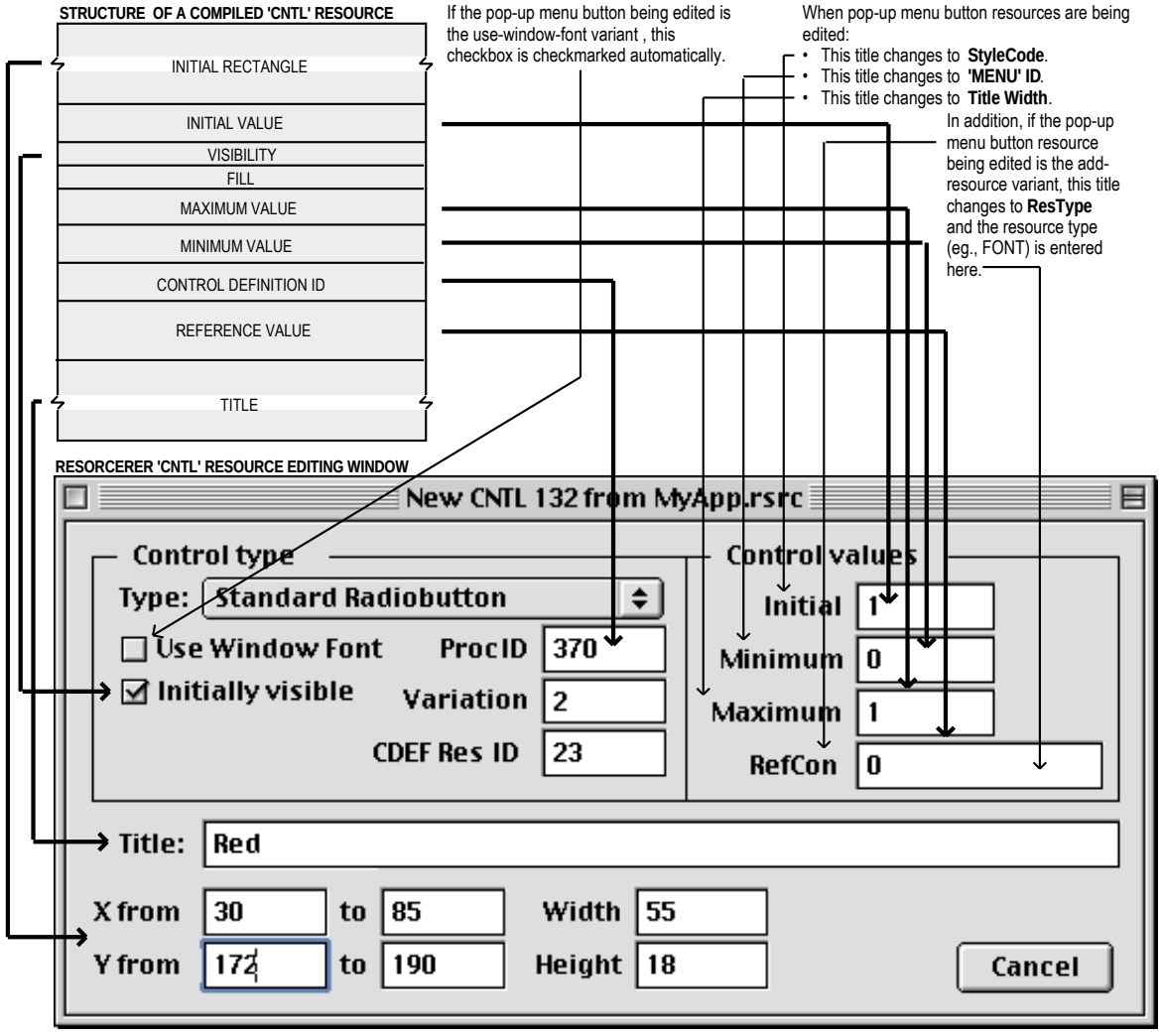

FIG 7 - CREATING A 'CNTL' RESOURCE USING RESORCERER

Creating Controls Programmatically

You can also create controls programmatically. The following functions, which were introduced with Carbon, may be used to programmatically create the controls described in this chapter:

Function	Parameters
CreatePushButtonControl	Rectangle, title.
CreatePushButtonWithIconControl	Rectangle, title, address of ControlButtonContentInfo structure, icon alignment.
CreateRadioButtonControl	Rectangle, title, auto-toggle/non-auto-toggle.
CreateCheckBoxControl	Rectangle, title, autotoggle/non-autotoggle.
CreateScrollBarControl	Rectangle, control value, minimum value,maximum value, size of view area, live-feedback/non-live-feedback, UPP to control action function.
CreatePopupButtonControl	Rectangle, title, menu ID,variable-width/non-variable-width, title width,title justification, title style.
CreateGroupBoxControl	Rectangle, title, primary/secondary.
CreateUserPaneControl	Rectangle, features.

Push Button Content and Icon Alignment

One of the parameters in calls to CreatePushButtonWithIconControl is the address of a structure of type ControlButtonContentInfo:

```
struct ControlButtonContentInfo
{
  ControlContentType contentType;
  union
  {
    SInt16      resID;
    CIconHandle cIconHandle;
    Handle      iconSuite;
    IconRef     iconRef;
    PicHandle   picture;
    Handle      ICONHandle;
  } u;
};
typedef struct ControlButtonContentInfo ControlButtonContentInfo;
```

If, for example, you wish to specify a colour icon for the button's icon content, you would assign kControlContentCIconRes to the contentType field and the icon's resource ID to the resID field.

Another parameter in calls to CreatePushButtonWithIconControl is a value of type ControlPushButtonIconAlignment. Relevant constants are:

```
kControlPushButtonIconOnLeft
kControlPushButtonIconOnRight
```

Additional Considerations — Scroll Bars

When creating scroll bars, you typically call GetNewControl or CreateScrollBarControl immediately after you create the window and 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.)

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.

A scroll bar for a document window is, by convention, 16 pixels wide (vertical scroll bars) and 16 pixels high (horizontal scroll bars). The Control Manager draws one-pixel lines around the scroll bar, based on the rectangle enclosing the scroll bar. As shown at Fig 8, these outside lines should overlap the inside lines of the window frame.

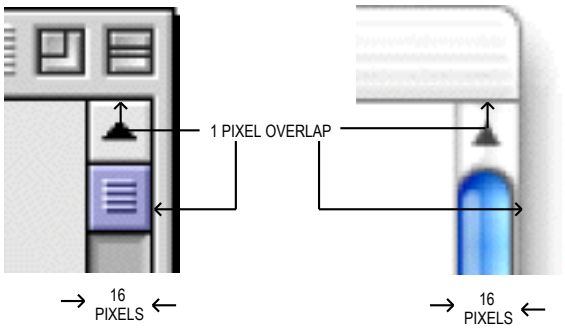

FIG 8 - CORRECT OVERLAP OF SCROLL BAR ON WINDOW FRAME

The following calculations[1] determine the rectangle for a vertical scroll bar for a document window:

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

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

When the user resizes the window, the initial maximum settings and location, as specified in the `'CNTL'` resource or `CreateScrollBarControl` call, must therefore be changed dynamically by the application as required. Typically, this is achieved by storing handles to each scroll bar in a document structure associated with the window and then using Control Manager functions to change control settings.

Additional Considerations – 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`	`0x0100`	Boldface font style
`popupTitleItalic`	`0x0200`	Italic font style
`popupTitleUnderline`	`0x0400`	Underline font style
`popupTitleOutline`	`0x0800`	Outline font style
`popupTitleShadow`	`0x1000`	Shadow font style
`popupTitleCondense`	`0x2000`	Condensed text
`popupTitleExtend`	`0x4000`	Extended text

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

Pop-up Menu Button Title Width

Fig 9 shows the relationship between the title width and the enclosing rectangle of a pop-up menu box.

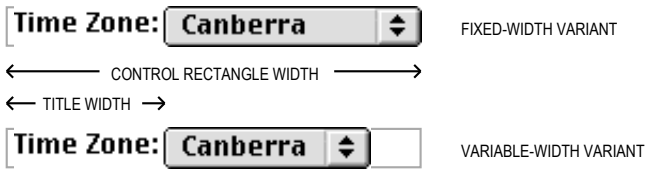

FIG 9 - POP-UP MENU BUTTON TITLE WIDTH AND CONTROL RECTANGLE WIDTH

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 9). 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 the control is created, the first menu item and the number of items in the pop-up menu button are stored in the control object. When the user chooses a different menu item, the Control Manager changes the item number value stored in the control object 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 reference constant value in the control object is coerced to type `ResType` and `AppendResMenu` is called 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. In this situation, you can still store a reference constant in the control object by calling `SetControlReference` after the control has been created.

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. 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

```
struct ControlFontStyleRec
{
  SInt16   flags;
  SInt16   font;
  SInt16   size;
  SInt16   style;
  SInt16   mode;
  SInt16   just;
  RGBColor foreColor;
  RGBColor backColor;
};
typedef struct ControlFontStyleRec ControlFontStyleRec;
typedef ControlFontStyleRec *ControlFontStylePtr;
```

Field Descriptions

`flags` Specifies which fields of the structure should be applied to the control (see Control Font Style Flag Constants, below). If no flags are set, the control uses the system font (or, for control variants which use the window font, the window font).

`font` Specifies the ID of the font family to use. Alternatively, a meta font constant may be assigned to this field (see Meta Font Constants, below).

`size` If the `kControlUseSizeMask` is set in the `flags` field, specifies the point size of the text. If the `kControlAddFontSizeMask` bit is set in the `flags` field, specifies the size to *add* to the current point size of the text. Alternatively, a meta font constant may be assigned to this field (see Meta Font Constants, below).

`style` Specifies the style to apply to the text. The bit numbers and the styles they represent are as follows:

Bit Number	Style
0	Bold
1	Italic
2	Underline
3	Outline
4	Shadow
5	Condensed
6	Extended

If all bits are clear, the plain font style is specified.

`mode` Specifies how characters are drawn in the bit image. (For a discussion of transfer modes, see Chapter 12.)

`just` Specifies text justification. The relevant constants are as follows:

Constant	Value
teFlushDefault	0
teCenter	1
teFlushRight	2
teFlushLeft	3

`foreColor` Specifies the RGB (red-green-blue) colour to use when drawing the text.

`backColor` Specifies the RGB 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 be applied to the control. If none of the flags in of 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
kControlUseFontMask	0x0001	The `font` field of the control font style structure is applied to the control.
kControlUseFaceMask	0x0002	The `style` field of the control font style structure is applied to the control. Ignored if you specify a meta font value (see Meta Font Constants, below).
kControlUseSizeMask	0x0004	The `size` field of the control font style structure is applied to the control. Ignored if you specify a meta font value (see Meta Font Constants, below).

kControlUseForeColorMask	0x0008	The foreColor field of the control font style structure is applied to the control. Applies only to static text controls.
kControlUseBackColorMask	0x0010	The backColor field of the control font style structure is applied to the control. Applies only to static text controls.
kControlUseModeMask	0x0020	The text mode specified in the mode field of the control font style structure is applied to the control.
kControlUseJustMask	0x0040	The just field of the control font style structure is applied to the control.
kControlUseAllMask	0x00FF	All flags in this mask will be set except kControlUseAddFontSizeMask.
kControlUseAddFontSizeMask	0x0100	The Dialog Manager will add a specified font size to the size field of the control font style structure. Ignored if you specify a meta font value (see Meta Font Constants, below.

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
kControlFontBigSystemFont	-1	Use the system font.
kControlFontSmallSystemFont	-2	Use the small system font.
kControlFontSmallBoldSystemFont	-3	Use the small emphasised system font.
kControlFontViewSystemFont	-4	Use the views font.

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.

Setting and Getting Control IDs

The following functions, which were introduced with Carbon, may be used to set and get a control's ID, and to get a reference to a control using the control's ID:

Function	Description
SetControlID	Set a control's ID.
GetControlID	Get a control's ID.
GetControlByID	Get a reference to a control using the control's ID.

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 alerts or dialogs, 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

DisposeControl may be used to remove a control from a window, delete it from the window's control list, and release the control object 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:
  - NULL for push buttons, checkboxes and radio buttons.
  - For scroll arrows and gray areas/tracks of scroll bars, a universal procedure 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/scroller of scroll bars:
    - NULL 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/scroller 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 control object.

  Note that, as an alternative to passing universal procedure pointers in the actionProc parameter of TrackControl, or the inAction parameter of HandleControlClick, you can preset the action function by passing the universal procedure pointer in the actionProc parameter of SetControlAction. (Ordinarily, you would call SetControlAction immediately after the control is created.) 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/track of a scroll bar, and move the scroll box/scroller accordingly.
  - If the non-live-feedback scroll bar variant is being used, and when the cursor is over the scroll box/scroller, determine where the user has dragged the scroll box/scroller, 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

`FindControl` will return both a reference to the control as well as a **control part code** when a mouse-down event occurs in a visible, active control. `FindControl` will set the control reference to `NULL` and return 0 as the control part code when a mouse-down occurs in an invisible or inactive control, or when the cursor is not in a control.

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 `HiliteControl` 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/scroller of a scroll bar.

A newer function (`FindControlUnderMouse`) will return a reference 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.

Tracking the Cursor in a Control

When the call to `FindControl` determines that the cursor was in a control when the user pressed the mouse button, you should call `TrackControl` or `HandleControlClick` to follow and respond to the user's movements.

You can use an action function to perform additional actions while 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/track of a scroll bar, or in the scroll box/scroller 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, and Getting a Control's Kind

Determining and Changing Control Settings

Your application often needs to determine the current setting and other values of a control when the user clicks the control. When the user clicks a non-auto-toggling checkbox, for example, your application must determine whether the box is currently checked before deciding whether to clear or draw the checkmark.

Your application can use the control value accessor functions described at The Control Object, above, to get and set a control's value, minimum value, and maximum value.

Getting a Control's Kind

You can determine a control's kind using `GetControlKind`. The control's kind is returned in a structure of type `ControlKind`:

```
struct ControlKind
{
  OSType signature;  // kControlKindSignatureApple ('appl') for all system controls
  OSType kind;
};
typedef struct ControlKind ControlKind;
```

For the controls addressed in this chapter, the following constants pertain to the `kind` field:

Constant	Value	Control Kind
`kControlKindPushButton`	`'push'`	Push button.
`kControlKindPushIconButton`	`'picn'`	Push button (colour icon variant).
`kControlKindCheckBox`	`'cbox'`	Checkbox.
`kControlKindRadioButton`	`'rdio'`	Radio button.
`kControlKindScrollBar`	`'sbar'`	Scroll bar.
`kControlKindPopupButton`	`'popb'`	Pop-up menu button.
`kControlKindGroupBox`	`'grpb'`	Primary group box (text title variant).
`kControlKindUserPane`	`'upan'`	User pane.

An alternative method is to call `GetControlData` with `kControlKindTag` passed in the `inTagName` parameter and the address of a variable of type `ControlKind` passed in the `inBuffer` parameter.

A problem here is that, as of the time of writing, both `GetControlKind` and the alternative method were only available on Mac OS X.

Moving and Resizing Scroll Bars

When the user resizes your windows, your application must resize and move that window's scroll bars. The steps involved are:

- Resize the window.
- Make each scroll bar invisible using `HideControl`.
- Move the scroll bars to the appropriate edges of the window using `MoveControl`.
- Lengthen or shorten each scroll bar (as appropriate) using `SizeControl`.
- After recalculating the maximum settings, update the settings and redraw the scroll boxes/scrollers using `SetControlMaximum` and `SetControlValue`.
- Make each scroll bar visible at its new location using `ShowControl`.

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

Scrolling Operations With Scroll Bars

Scrolling Basics

Relationship Between Document, Window, and Scroll Bar

The relationship between a document and a window, and their representation in a scroll bar, is shown at Fig 10.

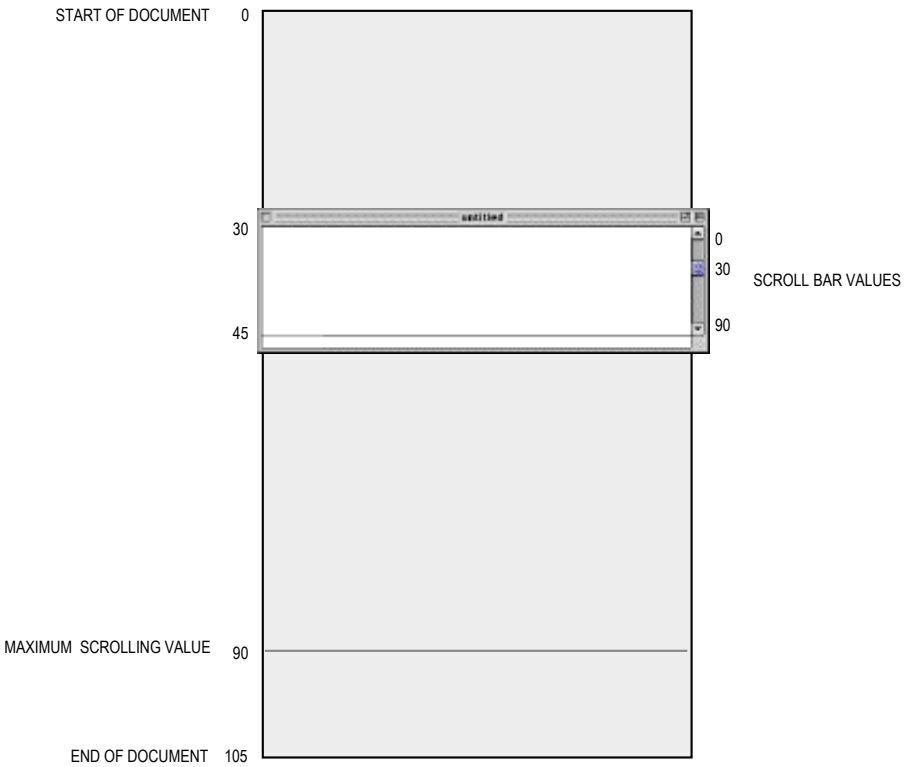

FIG 10 - RELATIONSHIP BETWEEN THE DOCUMENT, THE WINDOW, AND THE SCROLL BAR

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/scroller to a new location, your application should scroll the document a corresponding distance.
- For a click on a scroll arrow, you should scroll by a distance appropriate to your application. For example, a text editing application might scroll one line of text for one click in the vertical scroll bar.
- For a click in the gray area/track, you should scroll by a distance appropriate to your application. For example, a text editing application should ordinarily scroll by a distance equal to the height of the window less one text line.

Direction to scroll is determined by whether the scrolling distance is expressed as a positive or negative number. The scrolling distance will be expressed as a negative number if the user scrolls down from the beginning of the document, and as a positive number if the user scrolls up towards the beginning of the document.

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/Scroller

If the user did not effect the scroll using the scroll box/scroller, the scroll box/scroller 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 11. This newly opened document consists of 35 lines of text with a consistent line height of 10 pixels. The document is, therefore, 350 pixels long. The upper-left point of the document is identical to the window origin, that is, both are at (0,0).

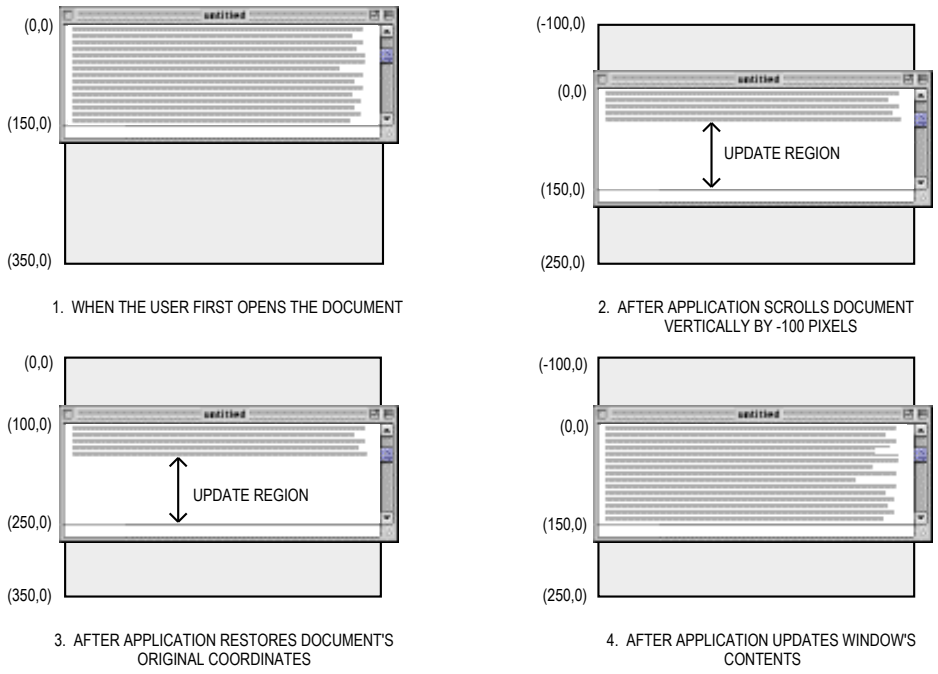

FIG 11 - SCROLLING A DOCUMENT IN A WINDOW

The window height is 150 pixels (15 lines of text). Thus the maximum setting for the scroll bar is equivalent to 200 pixels down in the document. (As shown at Fig 10, 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/scroller *down* the vertical scroll bar. This means that your application must move the text *up* by the appropriate amount. Moving a document up means a negative scrolling value.

Suppose the application, using `GetControlValue`, determines that the scroll bar's control value is 100. The application must then call `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 11, five lines from the bottom of the previous window display now appear at the top of the window.

Note that, in all this, `ScrollRect` does not change the coordinate system of the window; it simply moves the bits in the window to new coordinates that are still in the window's local coordinate system. 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 (see the update region at Fig 11), `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 11.)

The application now updates 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 11.)

To summarise:

- The user dragged the scroll box/scroller about half way down the vertical scroll bar. The application determined that this distance and direction to scroll was -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 21 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 22.)

Small Versions of Controls

Human Interface Guidelines permit the use of small versions of certain controls, which should only be used when space is at a premium. Full size and small controls should not be mixed in the same window.

The following lists those controls described in this chapter that are available in small versions, and describes how to create them.

Control	*Mac OS X*	*Mac OS 8/9*
Push button	Make the control's rectangle 17 pixels high and call `SetControlFontStyle` to set the small system font.	Make the control's rectangle 17 pixels high and use `SetControlFontStyle` to set the small system font.
Radio button Checkbox	Call `SetControlData` with the `kControlSizeTag` tag to make the control proper small, and call `SetControlFontStyle` to set the title to the small system font.	(Not available .)

Pop-up menu button	Create the control from a `'CNTL'` resource, specifying the `kControlPopupUseWFontVariant`, and set the owner window's font to the small system font.	Create the control from a `'CNTL'` resource, specifying the `kControlPopupUseWFontVariant`, and set the owner window's font to the small system font.
Scroll bar	Call `SetControlData` with the `kControlSizeTag` tag to make the control small.	Make the control's rectangle 11 pixels wide (vertical scroll bars) or high (horizontal scroll bars.

Small scroll bars should only be used in utility (floating) windows). Fig 12 shows an example.

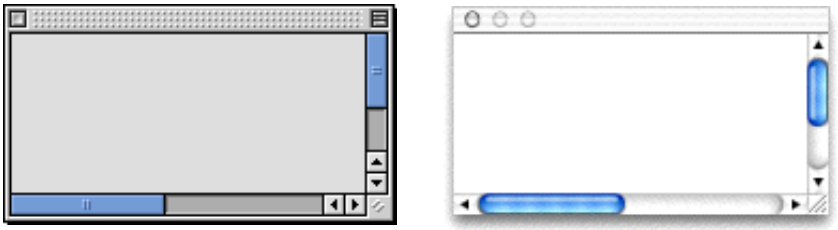

FIG 12 - SMALL SCROLL BARS IN A UTILITY (FLOATING) WINDOW

Main Control Manager Constants, Data Types and Functions Relevant to the Basic Controls, Primary Group Box & User Panes

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
```

Push Button Icon Alignment

```
kControlPushButtonIconOnLeft        = 6
kControlPushButtonIconOnRight       = 7
```

Pop-up Menu Button Variants

```
kControlPopupFixedWidthVariant      = 1 << 0
kControlPopupVariableWidthVariant   = 1 << 1
kControlPopupUseAddResMenuVariant   = 1 << 2
kControlPopupUseWFontVariant        = 1 << 3
```

Pop-up Title Characteristics

```
popupTitleBold                      = 1 << 8
popupTitleItalic                    = 1 << 9
popupTitleUnderline                 = 1 << 10
popupTitleOutline                   = 1 << 11
popupTitleShadow                    = 1 << 12
popupTitleCondense                  = 1 << 13
popupTitleExtend                    = 1 << 14
popupTitleNoStyle                   = 1 << 15
```

Control Variants

```
kControlNoVariant                   = 0
kControlUsesOwningWindowsFontVariant = 1 << 3
```

Control Part Codes

```
kControlNoPart                      = 0
kControlEntireControl               = 0
kControlLabelPart                   = 1
kControlMenuPart                    = 2
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')
kControlPopupButtonMenuRefTag        = FOUR_CHAR_CODE('mhan')
kControlPopupButtonMenuIDTag         = FOUR_CHAR_CODE('mnid')
kControlPopupButtonExtraHeightTag    = FOUR_CHAR_CODE('exht')
kControlGroupBoxTitleRectTag         = FOUR_CHAR_CODE('trec')
```

Control Data Tag Constants (Mac OS X Only)

```
kControlKindTag                      = FOUR_CHAR_CODE('kind')
kControlSizeTag                      = FOUR_CHAR_CODE('size')
```

Control Font Style Flag Constants

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

Meta Font Constants

```
kControlFontBigSystemFont            = -1
kControlFontSmallSystemFont          = -2
kControlFontSmallBoldSystemFont      = -3
kControlFontViewSystemFont           = -4
```

Push Button Icon Alignment

```
kControlPushButtonIconOnLeft         = 6
kControlPushButtonIconOnRight        = 7
```

Control Image Content

```
kControlContentIconSuiteRes          = 1
kControlContentCIconRes              = 2
kControlContentICONRes               = 4
kControlContentIconSuiteHandle       = 129
kControlContentCIconHandle           = 130
kControlContentPictHandle            = 131
kControlContentIconRef               = 132
kControlContentICON                  = 133
```

Control Kind (Mac OS X Only)

```
kControlKindPushButton               = 'push'
kControlKindPushIconButton           = 'picn'
kControlKindCheckBox                 = 'cbox'
kControlKindRadioButton              = 'rdio'
kControlKindScrollBar                = 'sbar'
kControlKindPopupButton              = 'popb'
kControlKindGroupBox                 = 'grpb'
kControlKindUserPane                 = 'upan'
```

Data Types

```
typedef struct OpaqueControlRef*  ControlRef;
typedef ControlRef  ControlHandle;
typedef SInt16      ControlPartCode;
typedef UInt16      ControlPushButtonIconAlignment;
typedef SInt16      ControlContentType;
```

Control Button Content Info

```
struct ControlButtonContentInfo
{
  ControlContentType  contentType;
  union
  {
    SInt16      resID;
    CIconHandle cIconHandle;
    Handle      iconSuite;
    IconRef     iconRef;
    PicHandle   picture;
    Handle      ICONHandle;
  } u;
};
typedef struct ControlButtonContentInfo ControlButtonContentInfo;
typedef ControlButtonContentInfo *ControlButtonContentInfoPtr;
```

Control Font Style

```
struct ControlFontStyleRec
{
  SInt16   flags;
  SInt16   font;
  SInt16   size;
  SInt16   style;
  SInt16   mode;
  SInt16   just;
  RGBColor foreColor;
  RGBColor backColor;
};
typedef struct ControlFontStyleRec ControlFontStyleRec;
typedef ControlFontStyleRec *ControlFontStylePtr;
```

Control ID

```
struct ControlID
{
  OSType signature;
  SInt32 id;
};
typedef struct ControlIDControlID;
```

ControlKind

```
struct ControlKind
{
  OSType signature;
  OSType kind;
};
typedef struct ControlKind ControlKind;
```

Functions

Creating Controls

```
ControlRef  GetNewControl(SInt16 controlID,WindowPtr owner);
ControlRef  NewControl(WindowPtr owningWindow,const Rect *boundsRect,ConstStr255Param
            title,Boolean initiallyVisible,SInt16 initialValue,SInt16 minimumValue,
            SInt16 maximumValue,SInt16 procID,SInt32 controlReference);
OSStatus    CreatePushButtonControl(WindowRef window,const Rect *boundsRect,
            ConstStr255Param title,ControlRef *outControl);
OSStatus    CreatePushButtonWithIconControl(WindowRef window,const Rect *boundsRect,
            ConstStr255Param title,ControlButtonContentInfo *icon,
            ControlPushButtonIconAlignment iconAlignment,ControlRef *outControl);
OSStatus    CreateRadioButtonControl(WindowRef window, const Rect *boundsRect,
            ConstStr255Param title,Boolean autoToggle,ControlRef *outControl);
OSStatus    CreateCheckBoxControl(WindowRef window,const Rect *boundsRect,
            ConstStr255Param title,Boolean autoToggle,ControlRef *outControl);
OSStatus    CreateScrollBarControl(WindowRef window,Rect *boundsRect,
            SInt32 value,SInt32 minimum,SInt32 maximum,SInt32 viewSize,
            Boolean liveTracking,ControlActionUPP liveTrackingProc,
```

```
            ControlRef *outControl);
OSStatus    CreatePopupButtonControl(WindowRef window,const Rect *boundsRect,
            ConstStr255Param title,SInt16 menuID,Boolean variableWidth,SInt16 titleWidth,
            SInt16 titleJustification,Style titleStyle,ControlRef *outControl);
OSStatus    CreateGroupBoxControl(WindowRef window,const Rect *boundsRect,
            ConstStr255Param title,Boolean primary,ControlRef *outControl);
OSStatus    CreateUserPaneControl(WindowRef window,const Rect *boundsRect,
            UInt32 features,ControlRef *outControl);
```

Removing Controls

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

Embedding Controls

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

Displaying and Manipulating Controls

```
void        MoveControl(ControlRef theControl,SInt16 h,SInt16 v);
void        SizeControl(ControlRef theControl,SInt16 w,SInt16 h);
void        UpdateControls(WindowPtr theWindow,RgnHandle updateRgn);
void        DrawControls(WindowPtr theWindow);
void        DrawOneControl(ControlRef theControl);
void        DrawControlInCurrentPort(ControlRef inControl);
Boolean     IsControlActive (ControlRef inControl);
OSErr       ActivateControl (ControlRef inControl);
OSErr       DeactivateControl(ControlRef inControl);
Boolean     IsControlVisible(ControlRef inControl);
void        HideControl(ControlRef theControl);
void        ShowControl(ControlRef theControl);
OSErr       SetControlVisibility (ControlRef inControl,Boolean inIsVisible,
            Boolean inDoDraw);
OSErr       SendControlMessage(ControlRef inControl,SInt16 inMessage,SInt32 inParam);
void        DragControl(ControlRef theControl,Point startPt,const Rect *limitRect,
            const Rect *slopRect,DragConstraint axis);
Boolean     IsControlHilited(ControlRef control);
void        HiliteControl(ControlRef theControl,ControlPartCode hiliteState);
```

Enabling and Disabling Controls (Mac OS X Only)

```
OSStatus    EnableControl(ControlRef inControl);
OSStatus    DisableControl(ControlRef inControl);
Boolean     IsControlEnabled(ControlRef inControl);
```

Handling Events in Controls

```
ControlPartCode  FindControl(Point thePoint,WindowPtr theWindow,ControlRef *theControl);
ControlRef       FindControlUnderMouse(Point inWhere,WindowPtr inWindow,
                 SInt16 *outPart);
ControlPartCode  TrackControl(ControlRef theControl,Point thePoint,
                 ControlActionUPP actionProc);
SInt16           HandleControlClick(ControlRef inControl,Point inWhere,SInt16 inModifiers,
                 ControlActionUPP nAction);
ControlPartCode  TestControl(ControlRef theControl, PointthePt);
```

Accessing and Changing Control Settings and Data

```
WindowPtr   GetControlOwner(ControlRef control);
void        SetControlOwner(ControlRef control,WindowPtr owner);
Rect *      GetControlBounds(ControlRef control,Rect *bounds);
void        SetControlBounds(ControlRef control,const Rect *bounds);
Boolean     IsControlHilited(ControlRef control);
UInt16      GetControlHilite(ControlRef control);
```

```
Handle      GetControlDefinition(ControlRef control);
Handle      GetControlDataHandle(ControlRef control);
void        SetControlDataHandle(ControlRef control,Handle dataHandle);
MenuHandle  GetControlPopupMenuHandle(ControlRef control);
void        SetControlPopupMenuHandle(ControlRef control,MenuHandle popupMenu);
short       GetControlPopupMenuID(ControlRef control);
void        SetControlPopupMenuID(ControlRef control,short menuID);
SInt16      GetControlValue(ControlRef theControl);
void        SetControlValue(ControlRef theControl,SInt16 theValue);
SInt16      GetControlMinimum(ControlRef theControl);
void        SetControlMinimum(ControlRef theControl,SInt16 newMinimum);
SInt16      GetControlMaximum(ControlRef theControl);
void        SetControlMaximum(ControlRef theControl,SInt16 newMaximum);
SInt32      GetControl32BitValue(ControlRef theControl);
void        SetControl32BitValue(ControlRef theControl,SInt32 newValue);
SInt32      GetControl32BitMaximum(ControlRef theControl);
void        SetControl32BitMaximum(ControlRef theControl,SInt32 newMaximum);
SInt32      GetControl32BitMinimum(ControlRef theControl);
void        SetControl32BitMinimum(ControlRef theControl,SInt32 newMinimum);
void        GetControlTitle(ControlRef theControl,Str255 title);
void        SetControlTitle(ControlRef theControl,ConstStr255Param title);
OSStatus    SetControlTitleWithCFString(ControlRef inControl,CFStringRef inString);
SInt32      GetControlReference(ControlRef theControl);
void        SetControlReference(ControlRef theControl,SInt32 data);
ControlActionUPP  GetControlAction(ControlRef theControl);
void        SetControlAction(ControlRef theControl,ControlActionUPP actionProc);
SInt32      GetControlViewSize(ControlRef theControl);
void        SetControlViewSize(ControlRef theControl,SInt32 newViewSize);
SInt16      GetControlVariant(ControlRef theControl);
OSErr       SetControlData(ControlRef inControl,ControlPartCode inPart,
            ResType inTagName,Size inSize,Ptr inData);
OSErr       GetControlData (ControlRef inControl,ControlPartCode inPart,
            ResType inTagName,Size inBufferSize,Ptr inBuffer,Size *outActualSize);
OSErr       GetControlDataSize(ControlRef inControl,ControlPartCode inPart,
            ResType inTagName,Size *outMaxSize);
OSErr       SetControlVisibility(ControlRef inControl,Boolean inIsVisible,
            Boolean inDoDraw);
```

Setting the Control Font Style

```
OSErr       SetControlFontStyle(ControlRef inControl,const ControlFontStyleRec *inStyle);
```

Setting and Getting Control IDs

```
OSStatus    SetControlID(ControlRef inControl,const ControlID *inID);
OSStatus    GetControlID(ControlRef inControl,ControlID *outID);
OSStatus    GetControlByID(WindowRef inWindow,const ControlID *inID,ControlRef *outControl);
```

Getting Control Kind (Mac OS X Only)

```
OSStatus    GetControlKind(ControlRef inControl,ControlKind *outControlKind);
```

Creating and Disposing of Universal Procedure Pointers for Control Action Functions

```
ControlActionUPP  NewControlActionUPP(ControlActionProcPtr userRoutine);
void        DisposeControlActionUPP(ControlActionUPP userUPP);
```

Application-Defined (Callback) Function

```
void        myControlActionFunction(ControlRef theControl,ControlPartCode partCode);
```

Demonstration Program Controls1 Listing

```
// ****************************************************************************************
// Controls1.c                                                          CLASSIC EVENT MODEL
// ****************************************************************************************
//
// This program opens a kWindowFullZoomGrowDocumentProc window containing:
//
// •  Three pop-up menu buttons (fixed width, variable width and use window font variants).
//
// •  Three non-auto-toggling radio buttons auto-embedded in a primary group box (text title
//    variant).
//
// •  Three non-auto-toggling 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).
//
// Some controls are created using 'CNTL' resources.  Others are created programmatically.
//
// 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/tracks of the scroll bars
//    are clicked and when the scroll box/scroller 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 contents.
//
// 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:
//
// •  A 'plst' resource.
//
// •  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 those controls not created programmatically.
//
// •  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,  canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// ****************************************************************************************

// ..... includes

#include <Carbon.h>
#include <string.h>

// ..... defines

#define rMenubar          128
```

```
#define rWindow           128
#define mAppleApplication 128
#define  iAbout           1
#define mFile             129
#define  iQuit            12
#define mDemonstration    131
#define  iColour          1
#define  iGrids           2
#define cPopupFixed       128
#define cPopupWinFont     129
#define cRadiobuttonRed   130
#define cRadiobuttonBlue  131
#define cCheckboxGrid     132
#define cCheckboxGridsnap 133
#define cGroupBoxColour   134
#define cButton           135
#define cButtonLeftIcon   136
#define cScrollbarVert    137
#define kScrollBarWidth   15
#define MAX_UINT32        0xFFFFFFFF
#define MIN(a,b)          ((a) < (b) ? (a) : (b))

// ..... typedefs

typedef struct
{
  ControlRef popupFixedRef;
  ControlRef popupVariableRef;
  ControlRef popupWinFontRef;
  ControlRef groupboxColourRef;
  ControlRef groupboxGridsRef;
  ControlRef buttonRef;
  ControlRef buttonDefaultRef;
  ControlRef buttonLeftIconRef;
  ControlRef buttonRightIconRef;
  ControlRef radiobuttonRedRef;
  ControlRef radiobuttonWhiteRef;
  ControlRef radiobuttonBlueRef;
  ControlRef checkboxGridRef;
  ControlRef checkboxRulersRef;
  ControlRef checkboxGridSnapRef;
  ControlRef scrollbarVertRef;
  ControlRef scrollbarHorizRef;
} docStruc;

typedef docStruc **docStrucHandle;

// ..... global variables

ControlActionUPP gActionFunctionVertUPP;
ControlActionUPP gActionFunctionHorizUPP;
Boolean          gRunningOnX = false;
WindowRef        gWindowRef;
Boolean          gDone;
Boolean          gInBackground = false;
Str255           gCurrentString;

// ..... function prototypes

void  main               (void);
void  doPreliminaries    (void);
OSErr quitAppEventHandler (AppleEvent *,AppleEvent *,SInt32);
void  doGetControls      (WindowRef);
void  doEvents           (EventRecord *);
void  doMouseDown        (EventRecord *);
void  doMenuChoice       (SInt32);
void  doUpdate           (EventRecord *);
void  doActivate         (EventRecord *);
void  doActivateWindow   (WindowRef,Boolean);
```

```
void  doOSEvent           (EventRecord *);
void  doInContent         (EventRecord *,WindowRef);
void  doPopupMenuChoice   (WindowRef,ControlRef,SInt16);
void  doVertScrollbar     (ControlPartCode,WindowRef,ControlRef,Point);
void  actionFunctionVert  (ControlRef,ControlPartCode);
void  actionFunctionHoriz (ControlRef,ControlPartCode);
void  doMoveScrollBox     (ControlRef,SInt16);
void  doRadioButtons      (ControlRef,WindowRef);
void  doCheckboxes        (ControlRef);
void  doPushButtons       (ControlRef,WindowRef);
void  doAdjustScrollBars  (WindowRef);
void  doDrawMessage       (WindowRef,Boolean);
void  doConcatPStrings    (Str255,Str255);
void  doCopyPString       (Str255,Str255);
void  helpTags            (WindowRef);

// ********************************************************************************** main

void  main(void)
{
  MenuBarHandle  menubarHdl;
  SInt32         response;
  MenuRef        menuRef;
  docStrucHandle docStrucHdl;
  EventRecord    EventStructure;

  // ..... do preliminaries

  doPreliminaries();

  // ..... create universal procedure pointers

  gActionFunctionVertUPP  = NewControlActionUPP((ControlActionProcPtr) actionFunctionVert);
  gActionFunctionHorizUPP = NewControlActionUPP((ControlActionProcPtr) actionFunctionHoriz);

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

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
      DisableMenuItem(menuRef,0);
    }

    gRunningOnX = true;
  }

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

  doCopyPString("\pBalloon and Help tag help is available",gCurrentString);

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

  if(!(gWindowRef = GetNewCWindow(rWindow,NULL,(WindowRef) -1)))
    ExitToShell();

  SetPortWindowPort(gWindowRef);
  UseThemeFont(kThemeSmallSystemFont,smSystemScript);
```

```
  SetThemeWindowBackground(gWindowRef,kThemeBrushDialogBackgroundActive,false);

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

  if(!(docStrucHdl = (docStrucHandle) NewHandle(sizeof(docStruc))))
    ExitToShell();

  SetWRefCon(gWindowRef,(SInt32) docStrucHdl);

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

  doGetControls(gWindowRef);
  doAdjustScrollBars(gWindowRef);

  if(gRunningOnX)
    helpTags(gWindowRef);

  ShowWindow(gWindowRef);

// ..... enter eventLoop

  gDone = false;

  while(!gDone)
  {
    if(WaitNextEvent(everyEvent,&EventStructure,MAX_UINT32,NULL))
      doEvents(&EventStructure);
  }
}

// ********************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  OSErr osError;

  MoreMasterPointers(128);
  InitCursor();
  FlushEvents(everyEvent,0);

  osError = AEInstallEventHandler(kCoreEventClass,kAEQuitApplication,
                            NewAEEventHandlerUPP((AEEventHandlerProcPtr) quitAppEventHandler),
                            0L,false);
  if(osError != noErr)
    ExitToShell();
}

// ********************************************************************** doQuitAppEvent

OSErr  quitAppEventHandler(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
  OSErr    osError;
  DescType returnedType;
  Size     actualSize;

  osError = AEGetAttributePtr(appEvent,keyMissedKeywordAttr,typeWildCard,&returnedType,NULL,0,
                              &actualSize);

  if(osError == errAEDescNotFound)
  {
    gDone = true;
    osError = noErr;
  }
  else if(osError == noErr)
    osError = errAEParamMissed;

  return osError;
}
```

```
// ************************************************************************** doGetControls

void  doGetControls(WindowRef windowRef)
{
  ControlRef               controlRef;
  docStrucHandle           docStrucHdl;
  OSStatus                 osError;
  Rect                     popupVariableRect    = { 73,  25, 93,245 };
  Rect                     radioButtonWhiteRect = { 183, 35,201, 92 };
  Rect                     checkboxRulersRect   = { 183,138,201,242 };
  Rect                     groupboxGridsRect    = { 136,123,236,252 };
  Rect                     buttonDefaultRect    = { 252,141,272,210 };
  Rect                     buttonRightIconRect  = { 285,141,305,220 };
  Rect                     scrollbarVertRect    = { 0,    0, 16,100 };
  ControlButtonContentInfo buttonContentInfo;
  Boolean                  booleanData = true;
  ControlFontStyleRec      controlFontStyleStruc;

  if(!gRunningOnX)
    CreateRootControl(windowRef,&controlRef);

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  // ..................................................................................................... popup menu buttons

  if(!((*docStrucHdl)->popupFixedRef = GetNewControl(cPopupFixed,windowRef)))
    ExitToShell();

  if((osError = CreatePopupButtonControl(windowRef,&popupVariableRect,CFSTR("Time Zone:"),
                                         132,true,76,popupTitleLeftJust,popupTitleNoStyle,
                                         &(*docStrucHdl)->popupVariableRef)) == noErr)
    ShowControl((*docStrucHdl)->popupVariableRef);
  else
    ExitToShell();

  if(!((*docStrucHdl)->popupWinFontRef = GetNewControl(cPopupWinFont,windowRef)))
    ExitToShell();

  // ..................................................................................................... radio buttons

  if(!((*docStrucHdl)->radiobuttonRedRef = GetNewControl(cRadiobuttonRed,windowRef)))
    ExitToShell();

  if((osError = CreateRadioButtonControl(windowRef,&radioButtonWhiteRect,CFSTR("White"),0,
                                        false,&(*docStrucHdl)->radiobuttonWhiteRef)) == noErr)
    ShowControl((*docStrucHdl)->radiobuttonWhiteRef);
  else
    ExitToShell();

  if(!((*docStrucHdl)->radiobuttonBlueRef = GetNewControl(cRadiobuttonBlue,windowRef)))
    ExitToShell();

  // ..................................................................................................... checkboxes

  if(!((*docStrucHdl)->checkboxGridRef = GetNewControl(cCheckboxGrid,windowRef)))
    ExitToShell();

  if((osError = CreateCheckBoxControl(windowRef,&checkboxRulersRect,CFSTR("Rulers"),0,false,
                                      &(*docStrucHdl)->checkboxRulersRef)) == noErr)
    ShowControl((*docStrucHdl)->checkboxRulersRef);
  else
    ExitToShell();

  if(!((*docStrucHdl)->checkboxGridSnapRef = GetNewControl(cCheckboxGridsnap,windowRef)))
    ExitToShell();

  // ..................................................................................................... group boxes

  if(!((*docStrucHdl)->groupboxColourRef = GetNewControl(cGroupBoxColour,windowRef)))
```

```
    ExitToShell();

  if((osError = CreateGroupBoxControl(windowRef,&groupboxGridsRect,CFSTR("Grids"),true,
                                      &(*docStrucHdl)->groupboxGridsRef)) == noErr)
    ShowControl((*docStrucHdl)->groupboxGridsRef);
  else
    ExitToShell();

  // ..... push buttons

  if(!((*docStrucHdl)->buttonRef = GetNewControl(cButton,windowRef)))
    ExitToShell();

  if((osError = CreatePushButtonControl(windowRef,&buttonDefaultRect,CFSTR("OK"),
                                        &(*docStrucHdl)->buttonDefaultRef)) == noErr)
    ShowControl((*docStrucHdl)->buttonDefaultRef);
  else
    ExitToShell();

  if(!((*docStrucHdl)->buttonLeftIconRef = GetNewControl(cButtonLeftIcon,windowRef)))
    ExitToShell();

  buttonContentInfo.contentType = kControlContentCIconRes;
  buttonContentInfo.u.resID = 101;
  if((osError = CreatePushButtonWithIconControl(windowRef,&buttonRightIconRect,
                                                CFSTR("Button"),&buttonContentInfo,
                                                kControlPushButtonIconOnRight,
                                                &(*docStrucHdl)->buttonRightIconRef)) == noErr)
    ShowControl((*docStrucHdl)->buttonRightIconRef);
  else
    ExitToShell();

  // ..... scroll bars

  if(!((*docStrucHdl)->scrollbarVertRef = GetNewControl(cScrollbarVert,windowRef)))
    ExitToShell();

  if((osError = CreateScrollBarControl(windowRef,&scrollbarVertRect,0,0,100,100,true,
                                       gActionFunctionHorizUPP,
                                       &(*docStrucHdl)->scrollbarHorizRef)) == noErr)
    ShowControl((*docStrucHdl)->scrollbarHorizRef);
  else
    ExitToShell();

  // .....

  AutoEmbedControl((*docStrucHdl)->radiobuttonRedRef,windowRef);
  AutoEmbedControl((*docStrucHdl)->radiobuttonWhiteRef,windowRef);
  AutoEmbedControl((*docStrucHdl)->radiobuttonBlueRef,windowRef);
  AutoEmbedControl((*docStrucHdl)->checkboxGridRef,windowRef);
  AutoEmbedControl((*docStrucHdl)->checkboxRulersRef,windowRef);
  AutoEmbedControl((*docStrucHdl)->checkboxGridSnapRef,windowRef);

  SetControlData((*docStrucHdl)->buttonDefaultRef,kControlEntireControl,
                 kControlPushButtonDefaultTag,sizeof(booleanData),&booleanData);

  controlFontStyleStruc.flags = kControlUseFontMask;
  controlFontStyleStruc.font = kControlFontSmallSystemFont;
  SetControlFontStyle((*docStrucHdl)->buttonLeftIconRef,&controlFontStyleStruc);
  controlFontStyleStruc.font = kControlFontSmallBoldSystemFont;
  SetControlFontStyle((*docStrucHdl)->buttonRightIconRef,&controlFontStyleStruc);

  DeactivateControl((*docStrucHdl)->checkboxRulersRef);
}

// ********************************************************************************* doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
```

```
  SInt32        menuChoice;
  MenuID        menuID;
  MenuItemIndex menuItem;

  switch(eventStrucPtr->what)
  {
    case kHighLevelEvent:
      AEProcessAppleEvent(eventStrucPtr);
      break;

    case keyDown:
      if((eventStrucPtr->modifiers & cmdKey) != 0)
      {
        menuChoice = MenuEvent(eventStrucPtr);
        menuID = HiWord(menuChoice);
        menuItem = LoWord(menuChoice);
        if(menuID == mFile && menuItem  == iQuit)
          gDone = true;
      }
      break;

    case mouseDown:
      doMouseDown(eventStrucPtr);
      break;

    case updateEvt:
      doUpdate(eventStrucPtr);
      break;

    case activateEvt:
      doActivate(eventStrucPtr);
      break;

    case osEvt:
      doOSEvent(eventStrucPtr);
      break;
  }
}

// ********************************************************************************** doMouseDown

void  doMouseDown(EventRecord *eventStrucPtr)
{
  WindowPartCode partCode, zoomPart;
  WindowRef      windowRef;
  Rect           constraintRect, mainScreenRect, portRect;
  BitMap         screenBits;
  Point          standardStateHeightAndWidth;

  partCode = FindWindow(eventStrucPtr->where,&windowRef);

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

    case inContent:
      if(windowRef != FrontWindow())
        SelectWindow(windowRef);
      else
        doInContent(eventStrucPtr,windowRef);
      break;

    case inDrag:
      DragWindow(windowRef,eventStrucPtr->where,NULL);
      break;

    case inGoAway:
```

```
      if(TrackGoAway(windowRef,eventStrucPtr->where) == true)
        gDone = true;
      break;

    case inGrow:
      constraintRect.top   = 341;
      constraintRect.left = 287;
      constraintRect.bottom = constraintRect.right = 32767;
      ResizeWindow(windowRef,eventStrucPtr->where,&constraintRect,NULL);
      doAdjustScrollBars(windowRef);
      doDrawMessage(windowRef,true);
      break;

    case inZoomIn:
    case inZoomOut:
      mainScreenRect = GetQDGlobalsScreenBits(&screenBits)->bounds;
      standardStateHeightAndWidth.v = mainScreenRect.bottom - 75;
      standardStateHeightAndWidth.h = 600;

      if(IsWindowInStandardState(windowRef,&standardStateHeightAndWidth,NULL))
        zoomPart = inZoomIn;
      else
        zoomPart = inZoomOut;

      if(TrackBox(windowRef,eventStrucPtr->where,partCode))
      {
        GetWindowPortBounds(windowRef,&portRect);
        EraseRect(&portRect);
        ZoomWindowIdeal(windowRef,zoomPart,&standardStateHeightAndWidth);
        doAdjustScrollBars(windowRef);
      }
      break;
  }
}

// ********************************************************************************* doMenuChoice

void  doMenuChoice(SInt32 menuChoice)
{
  MenuID         menuID;
  MenuItemIndex  menuItem;
  MenuRef        menuRef;
  WindowRef      windowRef;
  docStrucHandle docStrucHdl;

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

  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mAppleApplication:
      if(menuItem == iAbout)
        SysBeep(10);
      break;

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

    case mDemonstration:
      menuRef = GetMenuRef(mDemonstration);
      windowRef = FrontWindow();
      docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

      if(menuItem == iColour)
```

```
      {
        if(IsControlVisible((*docStrucHdl)->groupboxColourRef))
        {
          HideControl((*docStrucHdl)->groupboxColourRef);
          SetMenuItemText(menuRef,iColour,"\pShow Colour");
        }
        else
        {
          ShowControl((*docStrucHdl)->groupboxColourRef);
          SetMenuItemText(menuRef,iColour,"\pHide Colour");
        }
      }
      else if(menuItem == iGrids)
      {
        if(IsControlActive((*docStrucHdl)->groupboxGridsRef))
        {
          DeactivateControl((*docStrucHdl)->groupboxGridsRef);
          SetMenuItemText(menuRef,iGrids,"\pActivate Grids");
        }
        else
        {
          ActivateControl((*docStrucHdl)->groupboxGridsRef);
          SetMenuItemText(menuRef,iGrids,"\pDeactivate Grids");
        }
      }
      break;
  }

  HiliteMenu(0);
}

// ********************************************************************************** doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
  WindowRef windowRef;
  RgnHandle regionHdl;

  windowRef = (WindowRef) eventStrucPtr->message;

  BeginUpdate(windowRef);

  SetPortWindowPort(windowRef);
  doDrawMessage(windowRef,!gInBackground);

  if(regionHdl = NewRgn())
  {
    GetPortVisibleRegion(GetWindowPort(windowRef),regionHdl);
    UpdateControls(windowRef,regionHdl);
    DisposeRgn(regionHdl);
  }

  EndUpdate(windowRef);
}

// ********************************************************************************** doActivate

void  doActivate(EventRecord *eventStrucPtr)
{
  WindowRef windowRef;
  Boolean   becomingActive;

  windowRef = (WindowRef) eventStrucPtr->message;
  becomingActive = ((eventStrucPtr->modifiers & activeFlag) == activeFlag);
  doActivateWindow(windowRef,becomingActive);
}

// ********************************************************************** doActivateWindow
```

```
void  doActivateWindow(WindowRef windowRef,Boolean becomingActive)
{
  ControlRef controlRef;

  GetRootControl(windowRef,&controlRef);

  if(becomingActive)
  {
    ActivateControl(controlRef);
    doDrawMessage(windowRef,becomingActive);
  }
  else
  {
    DeactivateControl(controlRef);
    doDrawMessage(windowRef,becomingActive);
  }
}

// ********************************************************************************** doOSEvent

void  doOSEvent(EventRecord *eventStrucPtr)
{
  switch((eventStrucPtr->message >> 24) & 0x000000FF)
  {
    case suspendResumeMessage:
      if((eventStrucPtr->message & resumeFlag) == 1)
      {
        SetThemeCursor(kThemeArrowCursor);
        gInBackground = false;
      }
      else
        gInBackground = true;
      break;
  }
}

// ********************************************************************************* doInContent

void  doInContent(EventRecord *eventStrucPtr,WindowRef windowRef)
{
  docStrucHandle docStrucHdl;
  ControlRef     controlRef;
  SInt16         controlValue, controlPartCode;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  SetPortWindowPort(windowRef);
  GlobalToLocal(&eventStrucPtr->where);

  if(controlPartCode = FindControl(eventStrucPtr->where,windowRef,&controlRef))
  {
    if(controlRef == (*docStrucHdl)->popupFixedRef ||
       controlRef == (*docStrucHdl)->popupVariableRef ||
       controlRef == (*docStrucHdl)->popupWinFontRef)
    {
      TrackControl(controlRef,eventStrucPtr->where,(ControlActionUPP) -1);
      controlValue = GetControlValue(controlRef);
      doPopupMenuChoice(windowRef,controlRef,controlValue);
    }
    else if(controlRef == (*docStrucHdl)->scrollbarVertRef)
    {
      doVertScrollbar(controlPartCode,windowRef,controlRef,eventStrucPtr->where);
    }
    else if(controlRef == (*docStrucHdl)->scrollbarHorizRef)
    {
      TrackControl(controlRef,eventStrucPtr->where,gActionFunctionHorizUPP);
    }
    else
    {
```

```
      if(TrackControl(controlRef,eventStrucPtr->where,NULL))
      {
        if(controlRef == (*docStrucHdl)->radiobuttonRedRef ||
           controlRef == (*docStrucHdl)->radiobuttonWhiteRef ||
           controlRef == (*docStrucHdl)->radiobuttonBlueRef)
        {
          doRadioButtons(controlRef,windowRef);
        }
        if(controlRef == (*docStrucHdl)->checkboxGridRef ||
           controlRef == (*docStrucHdl)->checkboxRulersRef ||
           controlRef == (*docStrucHdl)->checkboxGridSnapRef)
        {
          doCheckboxes(controlRef);
        }
        if(controlRef == (*docStrucHdl)->buttonRef ||
           controlRef == (*docStrucHdl)->buttonDefaultRef ||
           controlRef == (*docStrucHdl)->buttonLeftIconRef ||
           controlRef == (*docStrucHdl)->buttonRightIconRef)
        {
          doPushButtons(controlRef,windowRef);
        }
      }
    }
  }
}

// ************************************************************************** doPopupMenuChoice

void  doPopupMenuChoice(WindowRef windowRef,ControlRef controlRef,SInt16 controlValue)
{
  MenuRef menuRef;
  Size    actualSize;

  GetControlData(controlRef,kControlEntireControl,kControlPopupButtonMenuHandleTag,
                 sizeof(menuRef),&menuRef,&actualSize);
  GetMenuItemText(menuRef,controlValue,gCurrentString);
  doDrawMessage(windowRef,true);
}

// ************************************************************************** doVertScrollbar

void  doVertScrollbar(ControlPartCode controlPartCode,WindowRef windowRef,
                      ControlRef controlRef,Point mouseXY)
{
  Str255 valueString;

  doCopyPString("\pScroll Bar Control Value: ",gCurrentString);

  switch(controlPartCode)
  {
    case kControlIndicatorPart:
      if(TrackControl(controlRef,mouseXY,NULL))
      {
        NumToString((SInt32) GetControlValue(controlRef),valueString);
        doConcatPStrings(gCurrentString,valueString);
        doDrawMessage(windowRef,true);
      }
      break;

    case kControlUpButtonPart:
    case kControlDownButtonPart:
    case kControlPageUpPart:
    case kControlPageDownPart:
      TrackControl(controlRef,mouseXY,gActionFunctionVertUPP);
      break;
  }
}

// ************************************************************************** actionFunctionVert
```

```
void  actionFunctionVert(ControlRef controlRef,ControlPartCode controlPartCode)
{
  SInt16    scrollDistance, controlValue;
  Str255    valueString;
  WindowRef windowRef;

  doCopyPString("\pScroll Bar Control Value: ",gCurrentString);

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

      case kControlPageUpPart:
      case kControlPageDownPart:
        scrollDistance = 55;
        break;
    }

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

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

    doMoveScrollBox(controlRef,scrollDistance);

    NumToString((SInt32) GetControlValue(controlRef),valueString);
    doConcatPStrings(gCurrentString,valueString);
    windowRef = GetControlOwner(controlRef);
    doDrawMessage(windowRef,true);
  }
}

// ********************************************************************** actionFunctionHoriz

void  actionFunctionHoriz(ControlRef controlRef,ControlPartCode controlPartCode)
{
  SInt16    scrollDistance, controlValue;
  Str255    valueString;
  WindowRef windowRef;

  doCopyPString("\pScroll Bar Control Value: ",gCurrentString);

  if(controlPartCode != kControlIndicatorPart)
  {
    switch(controlPartCode)
    {
      case kControlUpButtonPart:
      case kControlDownButtonPart:
        scrollDistance = 2;
        break;

      case kControlPageUpPart:
      case kControlPageDownPart:
        scrollDistance = 55;
        break;
    }

    if((controlPartCode == kControlDownButtonPart) ||
       (controlPartCode == kControlPageDownPart))
```

```
      scrollDistance = -scrollDistance;

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

    doMoveScrollBox(controlRef,scrollDistance);
  }

  NumToString((SInt32) GetControlValue(controlRef),valueString);
  doConcatPStrings(gCurrentString,valueString);
  windowRef = GetControlOwner(controlRef);
  doDrawMessage(windowRef,true);
}

// ********************************************************************** doMoveScrollBox

void doMoveScrollBox(ControlRef controlRef,SInt16 scrollDistance)
{
  SInt16 oldControlValue, controlValue, controlMax;

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

  controlValue = oldControlValue - scrollDistance;

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

  SetControlValue(controlRef,controlValue);
}

// ********************************************************************** doRadioButtons

void  doRadioButtons(ControlRef controlRef,WindowRef windowRef)
{
  docStrucHandle docStrucHdl;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  SetControlValue((*docStrucHdl)->radiobuttonRedRef,kControlRadioButtonUncheckedValue);
  SetControlValue((*docStrucHdl)->radiobuttonWhiteRef,kControlRadioButtonUncheckedValue);
  SetControlValue((*docStrucHdl)->radiobuttonBlueRef,kControlRadioButtonUncheckedValue);
  SetControlValue(controlRef,kControlRadioButtonCheckedValue);
}

// ********************************************************************** doCheckboxes

void  doCheckboxes(ControlRef controlRef)
{
  SetControlValue(controlRef,!GetControlValue(controlRef));
}

// ********************************************************************** doPushButtons

void  doPushButtons(ControlRef controlRef,WindowRef windowRef)
{
  docStrucHandle docStrucHdl;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  if(controlRef == (*docStrucHdl)->buttonRef)
  {
    doCopyPString("\pButton",gCurrentString);
    doDrawMessage(windowRef,true);
  }
```

```
  else if(controlRef == (*docStrucHdl)->buttonDefaultRef)
  {
    doCopyPString("\pDefault Button",gCurrentString);
    doDrawMessage(windowRef,true);
  }
  else if(controlRef == (*docStrucHdl)->buttonLeftIconRef)
  {
    doCopyPString("\pLeft Icon Button",gCurrentString);
    doDrawMessage(windowRef,true);
  }
  else if(controlRef == (*docStrucHdl)->buttonRightIconRef)
  {
    doCopyPString("\pRight Icon Button",gCurrentString);
    doDrawMessage(windowRef,true);
  }
}

// ********************************************************************** doAdjustScrollBars

void  doAdjustScrollBars(WindowRef windowRef)
{
  Rect           portRect;
  docStrucHandle docStrucHdl;

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  GetWindowPortBounds(windowRef,&portRect);

  HideControl((*docStrucHdl)->scrollbarVertRef);
  HideControl((*docStrucHdl)->scrollbarHorizRef);

  MoveControl((*docStrucHdl)->scrollbarVertRef,portRect.right - kScrollBarWidth,
              portRect.top + 25);
  MoveControl((*docStrucHdl)->scrollbarHorizRef,portRect.left -1,
              portRect.bottom - kScrollBarWidth);

  SizeControl((*docStrucHdl)->scrollbarVertRef,16, portRect.bottom - 39);
  SizeControl((*docStrucHdl)->scrollbarHorizRef, portRect.right - 13,16);

  ShowControl((*docStrucHdl)->scrollbarVertRef);
  ShowControl((*docStrucHdl)->scrollbarHorizRef);

  SetControlMaximum((*docStrucHdl)->scrollbarVertRef,portRect.bottom - portRect.top - 25
                    - kScrollBarWidth);
  SetControlMaximum((*docStrucHdl)->scrollbarHorizRef,portRect.right - portRect.left
                    - kScrollBarWidth);
}

// ************************************************************************** doDrawMessage

void  doDrawMessage(WindowRef windowRef,Boolean inState)
{
  Rect        portRect, headerRect;
  CFStringRef stringRef;
  Rect        textBoxRect;

  if(windowRef == gWindowRef)
  {
    SetPortWindowPort(windowRef);

    GetWindowPortBounds(windowRef,&portRect);

    SetRect(&headerRect,portRect.left - 1,portRect.top - 1,portRect.right + 1,
          portRect.top + 26);
    DrawThemeWindowHeader(&headerRect,inState);

    if(inState == kThemeStateActive)
      TextMode(srcOr);
    else
```

```
      TextMode(grayishTextOr);

    stringRef = CFStringCreateWithPascalString(NULL,gCurrentString,
                                               kCFStringEncodingMacRoman);
    SetRect(&textBoxRect,portRect.left,6,portRect.right,21);
    DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,true,&textBoxRect,teJustCenter,NULL);
    if(stringRef != NULL)
      CFRelease(stringRef);

    TextMode(srcOr);
  }
}

// ********************************************************************** doConcatPStrings

void  doConcatPStrings(Str255 targetString,Str255 appendString)
{
  SInt16 appendLength;

  appendLength = MIN(appendString[0],255 - targetString[0]);

  if(appendLength > 0)
  {
    BlockMoveData(appendString+1,targetString+targetString[0]+1,(SInt32) appendLength);
    targetString[0] += appendLength;
  }
}

// ************************************************************************** doCopyPString

void  doCopyPString(Str255 sourceString,Str255 destinationString)
{
  SInt16 stringLength;

  stringLength = sourceString[0];
  BlockMove(sourceString + 1,destinationString + 1,stringLength);
  destinationString[0] = stringLength;
}

// ****************************************************************************** helpTags

void  helpTags(WindowRef windowRef)
{
  docStrucHandle   docStrucHdl;
  HMHelpContentRec helpContent;

  memset(&helpContent,0,sizeof(helpContent));

  docStrucHdl = (docStrucHandle) (GetWRefCon(windowRef));

  HMSetTagDelay(500);
  HMSetHelpTagsDisplayed(true);
  helpContent.version = kMacHelpVersion;
  helpContent.tagSide = kHMOutsideTopCenterAligned;

  helpContent.content[kHMMinimumContentIndex].contentType = kHMStringResContent;
  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmResID = 128;

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 1;
  HMSetControlHelpContent((*docStrucHdl)->popupFixedRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 2;
  HMSetControlHelpContent((*docStrucHdl)->popupVariableRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 3;
  HMSetControlHelpContent((*docStrucHdl)->popupWinFontRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 4;
  HMSetControlHelpContent((*docStrucHdl)->groupboxColourRef,&helpContent);
```

```
  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 5;
  HMSetControlHelpContent((*docStrucHdl)->groupboxGridsRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 6;
  HMSetControlHelpContent((*docStrucHdl)->buttonRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 7;
  HMSetControlHelpContent((*docStrucHdl)->buttonDefaultRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 8;
  HMSetControlHelpContent((*docStrucHdl)->buttonLeftIconRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 9;
  HMSetControlHelpContent((*docStrucHdl)->buttonRightIconRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 10;
  HMSetControlHelpContent((*docStrucHdl)->scrollbarVertRef,&helpContent);

  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = 11;
  HMSetControlHelpContent((*docStrucHdl)->scrollbarHorizRef,&helpContent);
}

// ********************************************************************************************
```

Demonstration Program Controls1 Comments

When this program is run, the user should:

- On Mac OS 8/9, 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.
- On Mac OS X, peruse the Help tags which are invoked when the mouse cursor is held 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.
- 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/tracks of the scroll bars, noting the control value changes displayed in the window header.
- Drag the scroll box/scroller 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/scroller 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 hidden and shown 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.

In this program (and all others that use Help tags), the header file string.h is included and the library MSL C.PPC.Lib has been added to the project because of the use of memset in the function helpTags.

If you wish to display the Help tags, rather than Balloon help, on Mac OS 8/9:

- Comment out the line "if(gRunningOnX)" before the line "helpTags(gWindowRef);"
- In the function helpTags, change "kHMOutsideTopCenterAligned" to "kHMOutsideLeftCenterAligned". (At the time of writing, use of any but the first thirteen positioning constants in MacHelp.h defeated the display of Help tags on Mac OS 8/9.)

Help tag creation is addressed at Chapter 25.

defines

Constants are established for 'CNTL' resources for those controls not created programmatically.

typedef

The data type docStruc is a structure comprising fields in which the references to the control objects for the various controls will be stored. A handle to this structure will be stored in the window's window object.

Global Variables

actionFunctionVertUPP and actionFunctionHorizUPP will be assigned universal procedure pointers to action functions for the scroll bars.

main

Universal procedure pointers are created for the action functions for the two scroll bars.

The call to the function copyPString causes the string in the first parameter to be copied to the global variable gCurrentString. The string in 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 graphics port's font to the small system font. This latter is because one of the pop-up menus will use the window font. SetThemeWindowBackground is called to set the background colour/pattern for the window. The window's background will be similar to that applying to Mac OS 8/9 dialogs, which is appropriate for a window containing nothing but controls.

The call to NewHandle gets a relocatable block the size of one docStruc structure. The handle to the block is stored in the window's window object 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.

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

doGetControls

The function doGetControls creates the controls.

At the first line, if the program is running on Mac OS 8/9, CreateRootControl is called to create a root control for the window. On Mac OS 8/9, the first control created must be always be the root control (which is implemented as a user pane). This call is not necessary on Mac OS X because, on Mac OS X, root controls are created automatically for windows which have at least one control.

A handle to the structure in which the references to the control objects will be stored is then retrieved.

The controls are then created, some from 'CNTL' resources using GetNewControl and some programmatically. All of these calls create a control object for the relevant control and insert the object into the control list for the specified window. GetNewControl draws the controls created from 'CNTL' resources. In the case of controls created programmatically, ShowControl must me called to cause the control to be drawn. The reference to each control object is assigned to the appropriate field of the window's "document" structure.

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.

doMouseDown

doMouseDown switches 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 function doInContent is made.

The inGrow case is of particular significance to the scroll bars. Following the call to ResizeWindow, the function doAdjustScrollBars 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 call to doDrawMessage 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 window's port rectangle is erased before ZoomWindowIdeal zooms the window. Following the call to ZoomWindowIdeal the function doAdjustScrollBars is called to hide, move, resize, and redraw the scroll bars.

doMenuChoice

doMenuChoice handles user choices from the pull-down menus.

The mDemonstration case handles the Demonstration menu. Firstly, reference to that menu and a handle to the window's "document" structure 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 text is changed to "Show Colour". If it is not visible, ShowControl is called and the menu item text is changed to "Hide Colour".

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 (ignored on Mac OS X), 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 Mac OS 8/9 update region). The line preceding the if block is incidental to the demonstration. It simply redraws the window header frame and text in the window.

doActivateWindow

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

At the first line, GetRootControl gets a reference 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 global variable is then set to indicate whether the program has come to the foreground or has been sent to the background. This global variable is used in doUpdate, and controls the colour in which the text in the window header is drawn.

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 switching point for the further handling of those controls.

The first line gets the handle to the "document" structure containing the references to the various control objects. The call to SetPortWindowPort is a necessary precursor to the call to GlobalToLocal, which 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 newer function FindControlUnderMouse because there is no requirement to get a reference 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 reference returned by the FindControl call is first compared with the references 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 a function is called to perform further handling

Note that TrackControl, rather than the newer 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 reference returned by FindControl does not match the pop-up controls' references, it is then tested against the references to the vertical and horizontal scroll bar controls. If it matches the reference to the vertical scroll bar (which uses the non-live-feedback CDEF variant), the function doVertScrollbar is called to perform further handling. If it matches the reference to the horizontal scroll bar (which uses the live-feedback CDEF variant), TrackControl is called (ControlActionUPP) –1 passed in the actionProc parameter. This latter is because the UPP to the control action function has already been set. (Recall that it was passed in a parameter of the CreateScrollBarControl call which created the control.) The net effect of is that the application-defined action function to which the UPP relates will be repeatedly called while the mouse button remains down.

If the reference returned by FindControl does not match the references to any of the pop-up menu buttons or scroll bars, it must be a reference 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, NULL). If the cursor is still within the control when the mouse button is released, the reference is compared to, in sequence, the references to the radio buttons, the checkboxes, and the push buttons. If a match is found, the appropriate function is called to perform further handling.

doPopupMenuChoice

doPopupMenuChoice further handles mouse-downs in the pop-up menu buttons. In this demonstration, the further handling that would normally take place here is replaced by simply drawing the menu item text in the window.

The call to GetControlData gets a reference to the control's menu, allowing GetMenuItemText to get the item text into a global variable. This allows the text to be drawn 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 call to copyPString is incidental to the demonstration. It simply copies some appropriate text to the global variable gCurrentString.

At the next line, the function switches on the control part code. If the control part code was the scroll box/scroller (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/track 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.

ACTION FUNCTIONS

An action function is an example of a "callback function" (sometimes called a "hook function"). A callback function is an application-defined function that is called by a Toolbox function during the Toolbox function's execution, thus extending the features of the Toolbox function.

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/track).

The call to copyPString is incidental to the demonstration. It simply copies some appropriate text to the global variable gCurrentString.

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

The function switches 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/tracks.)

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 function doMoveScrollBox moves the scroll box/scroller the appropriate distance in the appropriate direction and updates the control's value accordingly. This means, of course, that the scroll box/scroller 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/scroller 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/scroller.

The call to copyPString is incidental to the demonstration. It simply copies some appropriate text to the global variable gCurrentString.

If the mouse-down occurred in the scroll box/scroller, 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 doMoveScrollBox to move the scroll box/scroller and update the control's value, is bypassed. The call to doMoveScrollBox is bypassed because, in live-feedback, the CDEF moves the scroll box/scroller and updates the control's value when the mouse-down is in the scroll box/scroller.

In this demonstration, the action taken after the main block of code has been bypassed (mouse-down in the scroll box/scroller) or executed (mouse-down in the scroll arrows or gray area/track) 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.)

doMoveScrollBox

doMoveScrollBox 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/scroller 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/scroller.

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 object.

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 causes 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.

Demonstration Program Controls2 Listing

```
// ****************************************************************************************
// Controls2.c                                                          CLASSIC EVENT MODEL
// ****************************************************************************************
//
// 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/scroller, the scroll arrows and the gray area/track 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.
//
// With regard to the scroll bars, the principal differences between this program and
// Controls1 are that, in this program:
//
// •  The scroll bar scroll boxes are made proportional.
//
// •  The action functions are set using the function SetControlAction.
//
// •  References to the scroll bar controls are not stored in, and retrieved from, a document
//    structure associated with the window.  Instead,  each control is assigned a controlID
//    using SetControlID, allowing the ID of the control to be retrieved using GetControlID
//    and a reference to the control to be obtained using GetControlByID.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  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).
//
// •  'PICT' resources containing the pictures to be scrolled (non-purgeable).
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// ****************************************************************************************

// ..... includes

#include <Carbon.h>

// ..... defines

#define rMenubar            128
#define rNewWindow          128
#define rPictureNonLive     128
#define rPictureLive        129
#define mAppleApplication   128
#define  iAbout             1
#define mFile               129
#define  iQuit              12
#define cScrollbarNonLive   128
#define cScrollbarLive      129
#define kScrollbarNonLiveID 1
#define kScrollbarLiveID    2
#define MAX_UINT32          0xFFFFFFFF

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

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

// ..... function prototypes

void  main                (void);
void  doPreliminaries     (void);
OSErr quitAppEventHandler (AppleEvent *,AppleEvent *,SInt32);
void  doEvents            (EventRecord *);
void  doMouseDown         (EventRecord *);
void  doUpdate            (EventRecord *);
void  doActivate          (EventRecord *);
void  doActivateWindow    (WindowRef,Boolean);
void  doOSEvent           (EventRecord *);
void  doMenuChoice        (SInt32);
void  doInContent         (EventRecord *,WindowRef);
void  doNonLiveScrollBars (ControlPartCode,WindowRef,ControlRef,Point);
void  actionFuncNonLive   (ControlRef,ControlPartCode);
void  actionFuncLive      (ControlRef,ControlPartCode);
void  doMoveScrollBox     (ControlRef,SInt16);

// ********************************************************************************** main

void  main(void)
{
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;
  WindowRef     windowRef;
  ControlRef    controlRefScrollbarNonLive, controlRefScrollbarLive;
  ControlID     controlID;
  Rect          portRect;
  EventRecord   eventStructure;

  // ..... do preliminaries

  doPreliminaries();

  // ..... create universal procedure pointers

  gActionFuncNonLiveUPP = NewControlActionUPP((ControlActionProcPtr) actionFuncNonLive);
  gActionFuncLiveUPP    = NewControlActionUPP((ControlActionProcPtr) actionFuncLive);

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

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
      DisableMenuItem(menuRef,0);
    }
  }

  // ..... open a window
```

```
  if(!(windowRef = GetNewCWindow(rNewWindow,NULL,(WindowRef)-1)))
    ExitToShell();

  SetPortWindowPort(windowRef);

  // ..... get controls and set ID and control action functions

  controlRefScrollbarNonLive  = GetNewControl(cScrollbarNonLive,windowRef);
  controlID.signature = 'kjB ';
  controlID.id = kScrollbarNonLiveID;
  SetControlID(controlRefScrollbarNonLive,&controlID);

  SetControlAction(controlRefScrollbarNonLive,gActionFuncNonLiveUPP);

  controlRefScrollbarLive = GetNewControl(cScrollbarLive,windowRef);
  controlID.id = kScrollbarLiveID;
  SetControlID(controlRefScrollbarLive,&controlID);

  SetControlAction(controlRefScrollbarLive,gActionFuncLiveUPP);

  // ..... get picture

  if(!(gPictHandleNonLive = GetPicture(rPictureNonLive)))
    ExitToShell();
  gPictRectNonLive = (*gPictHandleNonLive)->picFrame;

  if(!(gPictHandleLive = GetPicture(rPictureLive)))
    ExitToShell();
  gPictRectLive = (*gPictHandleLive)->picFrame;
  OffsetRect(&gPictRectLive,0,191);

  // ..... set up for proportional scroll boxes

  GetWindowPortBounds(windowRef,&portRect);
  SetControlViewSize(controlRefScrollbarNonLive,portRect.right);
  SetControlViewSize(controlRefScrollbarLive,portRect.right);

  // ..... enter eventLoop

  gDone = false;

  while(!gDone)
  {
    if(WaitNextEvent(everyEvent,&eventStructure,MAX_UINT32,NULL))
      doEvents(&eventStructure);
  }
}

// ********************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  OSErr osError;

  MoreMasterPointers(32);
  InitCursor();
  FlushEvents(everyEvent,0);

  osError = AEInstallEventHandler(kCoreEventClass,kAEQuitApplication,
                            NewAEEventHandlerUPP((AEEventHandlerProcPtr) quitAppEventHandler),
                            0L,false);
  if(osError != noErr)
    ExitToShell();
}

// ********************************************************************** doQuitAppEvent

OSErr  quitAppEventHandler(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
```

```
  OSErr    osError;
  DescType returnedType;
  Size     actualSize;

  osError = AEGetAttributePtr(appEvent,keyMissedKeywordAttr,typeWildCard,&returnedType,NULL,0,
                              &actualSize);

  if(osError == errAEDescNotFound)
  {
    gDone = true;
    osError = noErr;
  }
  else if(osError == noErr)
    osError = errAEParamMissed;

  return osError;
}

// ********************************************************************************** doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
  switch(eventStrucPtr->what)
  {
    case kHighLevelEvent:
      AEProcessAppleEvent(eventStrucPtr);
      break;

    case mouseDown:
      doMouseDown(eventStrucPtr);
      break;

    case updateEvt:
      doUpdate(eventStrucPtr);
      break;

    case activateEvt:
      doActivate(eventStrucPtr);
      break;

    case osEvt:
      doOSEvent(eventStrucPtr);
      break;
  }
}

// ********************************************************************************* doMouseDown

void  doMouseDown(EventRecord *eventStrucPtr)
{
  WindowRef      windowRef;
  WindowPartCode partCode;

  partCode = FindWindow(eventStrucPtr->where,&windowRef);

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

    case inContent:
      if(windowRef != FrontWindow())
        SelectWindow(windowRef);
      else
        doInContent(eventStrucPtr,windowRef);
      break;

    case inDrag:
```

```
      DragWindow(windowRef,eventStrucPtr->where,NULL);
      break;
  }
}

// ********************************************************************************** doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
  WindowRef  windowRef;
  RgnHandle  regionHdl;
  ControlID  controlID;
  ControlRef controlRef;

  windowRef = (WindowRef) eventStrucPtr->message;

  BeginUpdate(windowRef);

  SetPortWindowPort(windowRef);

  regionHdl = NewRgn();
  if(regionHdl)
  {
    GetPortVisibleRegion(GetWindowPort(windowRef),regionHdl);
    UpdateControls(windowRef,regionHdl);
    DisposeRgn(regionHdl);
  }

  controlID.signature = 'kjB ';
  controlID.id = kScrollbarNonLiveID;
  GetControlByID(windowRef,&controlID,&controlRef);
  SetOrigin(GetControlValue(controlRef),0);
  DrawPicture(gPictHandleNonLive,&gPictRectNonLive);
  SetOrigin(0,0);

  controlID.id = kScrollbarLiveID;
  GetControlByID(windowRef,&controlID,&controlRef);
  SetOrigin(GetControlValue(controlRef),0);
  DrawPicture(gPictHandleLive,&gPictRectLive);
  SetOrigin(0,0);

  EndUpdate(windowRef);
}

// ********************************************************************************** doActivate

void  doActivate(EventRecord *eventStrucPtr)
{
  WindowRef windowRef;
  Boolean   becomingActive;

  windowRef = (WindowRef) eventStrucPtr->message;
  becomingActive = ((eventStrucPtr->modifiers & activeFlag) == activeFlag);
  doActivateWindow(windowRef,becomingActive);
}

// ********************************************************************** doActivateWindow

void  doActivateWindow(WindowRef windowRef,Boolean becomingActive)
{
  ControlID  controlID;
  ControlRef controlRefScrollbarNonLive, controlRefScrollbarLive;

  controlID.signature = 'kjB ';
  controlID.id = kScrollbarNonLiveID;
  GetControlByID(windowRef,&controlID,&controlRefScrollbarNonLive);
  controlID.id = kScrollbarLiveID;
  GetControlByID(windowRef,&controlID,&controlRefScrollbarLive);
```

```
  if(becomingActive)
  {
    ActivateControl(controlRefScrollbarNonLive);
    ActivateControl(controlRefScrollbarLive);
  }
  else
  {
    DeactivateControl(controlRefScrollbarNonLive);
    DeactivateControl(controlRefScrollbarLive);
  }
}

// ********************************************************************************** doOSEvent

void  doOSEvent(EventRecord *eventStrucPtr)
{
  switch((eventStrucPtr->message >> 24) & 0x000000FF)
  {
    case suspendResumeMessage:
      if((eventStrucPtr->message & resumeFlag) == 1)
        SetThemeCursor(kThemeArrowCursor);
      break;
  }
}

// ********************************************************************************** doMenuChoice

void  doMenuChoice(SInt32 menuChoice)
{
  MenuID        menuID;
  MenuItemIndex menuItem;

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

  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mAppleApplication:
      if(menuItem == iAbout)
        SysBeep(10);
      break;

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

  HiliteMenu(0);
}

// ********************************************************************************** doIncontent

void  doInContent(EventRecord *eventStrucPtr,WindowRef windowRef)
{
  ControlPartCode controlPartCode;
  ControlRef      controlRef;
  ControlID       controlID;

  SetPortWindowPort(windowRef);
  GlobalToLocal(&eventStrucPtr->where);

  if(controlPartCode = FindControl(eventStrucPtr->where,windowRef,&controlRef))
  {
    GetControlID(controlRef,&controlID);
```

```
    if(controlID.id == kScrollbarNonLiveID)
      doNonLiveScrollBars(controlPartCode,windowRef,controlRef,eventStrucPtr->where);
    else if(controlID.id == kScrollbarLiveID)
      TrackControl(controlRef,eventStrucPtr->where,(ControlActionUPP) -1);
  }
}

// ********************************************************************** doNonLiveScrollBars

void  doNonLiveScrollBars(ControlPartCode controlPartCode,WindowRef windowRef,
                          ControlRef controlRef,Point mouseXY)
{
  SInt16    oldControlValue;
  SInt16    scrollDistance;
  RgnHandle updateRgnHdl;

  switch(controlPartCode)
  {
    case kControlIndicatorPart:
      oldControlValue = GetControlValue(controlRef);
      if(TrackControl(controlRef,mouseXY,NULL))
      {
        scrollDistance = oldControlValue - GetControlValue(controlRef);
        if(scrollDistance != 0)
        {
          updateRgnHdl = NewRgn();
          ScrollRect(&gPictRectNonLive,scrollDistance,0,updateRgnHdl);
          InvalWindowRgn(windowRef,updateRgnHdl);
          DisposeRgn(updateRgnHdl);
        }
      }
      break;

    case kControlUpButtonPart:
    case kControlDownButtonPart:
    case kControlPageUpPart:
    case kControlPageDownPart:
      TrackControl(controlRef,mouseXY,(ControlActionUPP) -1);
      break;
  }
}

// ********************************************************************** actionFuncNonLive

void  actionFuncNonLive(ControlRef controlRef,ControlPartCode controlPartCode)
{
  WindowRef windowRef;
  SInt16    scrollDistance, controlValue;
  Rect      portRect;
  RgnHandle updateRgnHdl;

  if(controlPartCode)
  {
    windowRef = GetControlOwner(controlRef);

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

      case kControlPageUpPart:
      case kControlPageDownPart:
        GetWindowPortBounds(windowRef,&portRect);
        scrollDistance = (portRect.right - portRect.left - 10);
        break;
    }
```

```
    if((controlPartCode == kControlDownButtonPart) ||
       (controlPartCode == kControlPageDownPart))
      scrollDistance = -scrollDistance;

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

    doMoveScrollBox(controlRef,scrollDistance);

    if(controlPartCode == kControlUpButtonPart ||
       controlPartCode == kControlDownButtonPart)
    {
      updateRgnHdl = NewRgn();
      ScrollRect(&gPictRectNonLive,scrollDistance,0,updateRgnHdl);
      InvalWindowRgn(windowRef,updateRgnHdl);
      DisposeRgn(updateRgnHdl);
      BeginUpdate(windowRef);
    }

    SetOrigin(GetControlValue(controlRef),0);
    DrawPicture(gPictHandleNonLive,&gPictRectNonLive);
    SetOrigin(0,0);

    if(controlPartCode == kControlUpButtonPart || controlPartCode == kControlDownButtonPart)
      EndUpdate(windowRef);
  }
}

// ************************************************************************** actionFuncLive

void  actionFuncLive(ControlRef controlRef,ControlPartCode partCode)
{
  WindowRef windowRef;
  SInt16    scrollDistance, controlValue;
  Rect      portRect;

  windowRef = GetControlOwner(controlRef);

  if(partCode != 0)
  {
    if(partCode != kControlIndicatorPart)
    {
      switch(partCode)
      {
        case kControlUpButtonPart:
        case kControlDownButtonPart:
          scrollDistance = 2;
          break;

        case kControlPageUpPart:
        case kControlPageDownPart:
          GetWindowPortBounds(windowRef,&portRect);
          scrollDistance = (portRect.right - portRect.left) - 10;
          break;
      }

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

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

      doMoveScrollBox(controlRef,scrollDistance);
    }
```

```
    SetOrigin(GetControlValue(controlRef),0);
    DrawPicture(gPictHandleLive,&gPictRectLive);
    SetOrigin(0,0);
  }
}

// ********************************************************************** doMoveScrollBox

void doMoveScrollBox(ControlRef controlRef,SInt16 scrollDistance)
{
  SInt16 oldControlValue, controlValue, controlMax;

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

  controlValue = oldControlValue - scrollDistance;

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

  SetControlValue(controlRef,controlValue);
}

// ********************************************************************************************
```

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. Also, this program supports proportional scroll boxes/scrollers.

When the program is run, the user should scroll the pictures by dragging the scroll boxes/scrollers, clicking in the scroll bar gray areas/tracks, clicking in the scroll arrows and holding the mouse button down while the cursor is in the gray areas/travks and scroll arrows. The user should note, when scrolling with the scroll boxes/scrollers, 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, respectively 1220 by 175 pixels and 915 by 175 pixels. "pane" for each picture and scroll bar is 400 pixels wide by 175 pixels high, the 'CNTL' resources set the control maximum values to 820 and 515 respectively, 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 caused by the way in which the image is drawn by the monitor on its screen.

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

defines

kScrollbarNonLiveID and kScrollbarLiveID will be assigned as the IDs of, respectively, the top (non-live) scroll bar and the bottom (live) scroll bar.

main

Two calls to GetNewControl allocate memory for the control objects, insert the control objects into the window's control list and draw the controls.

Following each call to GetNewControl:

- The id field of a variable of type ControlID is assigned the appropriate ID value for the specific control and the ID is then assigned to the control by a call to SetControlID.

- SetControlAction is called with a 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. As a consequence of using SetControlAction, (ControlActionUPP) -1 will be 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 191 pixels. (Note that the two 'PICT' resources were created so that the top and left fields of the picFrame Rect are both zero.)

In the next block, the width of the port rectangle is passed in the newViewSize parameter of calls to SetControlViewSize. (In this case, the view width is the same as the width of the port rectangle. 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 (on Mac OS 8/9, provided that the user has selected Smart Scrolling on in the Options tab of the Appearance control panel).

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).

Note that GetControlByID is used to retrieve references to the two controls.

DoActivateWindow

Note that GetControlByID is used to retrieve references to the two controls.

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.

If the mouse-down was in a scroll bar, GetControlID is called to get the ID of the control. Then, as in the demonstration program Controls1:

- If the mouse-down was in the non-live-feedback scroll bar, one of the application's functions 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 (callback) function associated with the UPP previously set by SetControlAction 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 switches on the control part code:

- If the mouse-down was in the scroll box/scroller (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 scroll box/scroller, 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/scroller, 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 (Mac OS 8/9). 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/track, more specifically in one of the non-live-feedback's scroll bar's scroll arrows or gray areas/tracks, TrackControl takes control until the user releases the mouse button. The third parameter in the TrackControl call means that the application-defined (callback) function associated with the UPP set by SetControlAction 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/track, the action function exits. The following occurs only when the cursor is within the control.

A reference to the window which "owns" this control is retrieved from the control object.

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/track, 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 function doMoveScrollBox adds/subtracts the distance to scroll to/from the control's current value and repositions the scroll box/scroller 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 InvalWindowRgn, and BeginUpdate is called to ensure that, on Mac OS 8/9, (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/track, but also while the scroll box/scroller 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/scroller.

If the mouse-down occurred in the scroll box/scroller, 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 doMoveScrollBox to move the scroll box/scroller and update the control's value, is bypassed. The call to doMoveScrollBox is bypassed because, the live-feedback variant of the CDEF moves the scroll box/scroller and updates the control's value when the mouse-down is in the scroll box/scroller.

After the if block has been bypassed (mouse-down in the scroll box/scroller) or executed (mouse-down in the scroll arrows or gray area/track), the window contents are scrolled. Regardless of whether the picture is being scrolled using the scroll box/scroller, the scroll arrows, or the gray areas/track, 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.

doMoveScrollBox

doMoveScrollBox 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.