You can use Control Manager routines to change values in the control record, or you can access and change its fields yourself; normally, you don't change the values in the auxiliary control record. However, both the control record and the auxiliary control record have fields in which your application can store information as you deem appropriate.

You can obtain the menu handle and the menu ID of the menu associated with a pop-up menu by dereferencing the `contrlData` field of the control record, which, for pop-up menu controls, contains a handle to a pop-up private data record. This record contains the menu handle and the menu ID for the associated menu.

You use a control color table record only when you want to use nonstandard colors for a control that you create while your application is running. Your application probably shouldn't ever create a control color table record because you should use the system's default colors to ensure consistency of the interface across applications.

The Control Record

When you create a control, the Control Manager incorporates the information you specify (either in the control resource or in the parameters of the `NewControl` function) into a control record, which is a data structure of type `ControlRecord`. The Control Manager functions you use for creating a control, `GetNewControl` and `NewControl`, return a handle to a newly allocated control record. Thereafter, your application normally refers to the control by this handle, because most other Control Manager routines expect a control handle as their first parameter.

You can use Control Manager routines to determine and change several of the values in the control record, or you can access and change its fields yourself.

```
TYPE ControlRecord =
PACKED RECORD
   nextControl:   ControlHandle; {next control}
   contrlOwner:   WindowPtr;     {control's window}
   contrlRect:    Rect;          {rectangle}
   contrlVis:     Byte;          {255 if visible}
   contrlHilite:  Byte;          {highlight state}
   contrlValue:   Integer;       {control's current setting}
   contrlMin:     Integer;       {control's minimum setting}
   contrlMax:     Integer;       {control's maximum setting}
   contrlDefProc: Handle;        {control definition function}
   contrlData:    Handle;        {data used by contrlDefProc}
   contrlAction:  ProcPtr;       {action procedure}
   contrlRfCon:   LongInt;       {control's reference value}
   contrlTitle:   Str255;        {control's title}
END;
```

Field descriptions

`nextControl` A handle to the next control associated with this control's window. All the controls belonging to a given window are kept in a linked list, beginning in the `controlList` field of the window record and chained together through the `nextControl` fields of the individual control records. The end of the list is marked by a `NIL` value; as new controls are created, they're added to the beginning of the list.

`contrlOwner` A pointer to the window to which this control belongs.

`contrlRect` The rectangle that completely encloses the control, in the local coordinates of the control's window. You can use the `MoveControl` and `SizeControl` procedures to change the rectangle stored in this field.

`contrlVis` The invisible/visible state for the control. When the value of this field is 0, the Control Manager does not draw the control (its state is invisible); when the value of this field is 255, the Control Manager draws the control (its state is visible). Note that even when a control is visible, it might still be obscured from sight by an overlapping window or some other object. You can use the `HideControl` procedure to change this field from visible to invisible, and you can use the `ShowControl` procedure to change this field from invisible to visible.

`contrlHilite` Specifies whether and how the control is to be displayed, indicating whether it's active or inactive and, if active, whether it's selected. The value of 0 signifies an active control that is not selected. A value from 1 through 253 signifies a part code designating the part of the (active) control to highlight, indicating that the user is pressing the mouse button while the cursor is in that part. The value 255 signifies that the control is to be made inactive and drawn accordingly. The `HiliteControl` procedure lets you change the value of this field.

`contrlValue` The control's current setting. For buttons, checkboxes, and radio buttons, 0 means the control is off and 1 means it's on. For scroll bars and other sliders, `contrlValue` may take any value within the range specified in the `contrlMin` and `contrlMax` fields. For pop-up menus, this value is the item number of the menu item chosen by the user; if the user hasn't chosen a menu item, it is the item number of the first menu item. For other controls, you can use this field as you wish. You can use the `GetControlValue` function to determine the value of this field, and you can use the `SetControlValue` procedure to change the value of this field.

`contrlMin` The control's minimum possible setting. For on-and-off controls—like checkboxes and radio buttons—this value should be 0 (meaning that the control is off). For scroll bars and other sliders, this can be any appropriate minimum value. For controls—like buttons—that don't retain a setting, this value should be 0. For pop-up menus, the Control Manager sets this field to 1. For other controls, you can use this field as you wish. You can use the `GetControlMinimum` function to determine the value of this field, and you can use the `SetControlMinimum` procedure to change the value of this field.

`contrlMax` The control's maximum possible setting. For on-and-off controls like checkboxes and radio buttons, this value should be 1 (meaning that the control is on). For scroll bars and other sliders, this can be any appropriate maximum value. When you make the maximum setting of a scroll bar equal to its minimum setting, the control definition function automatically makes the scroll bar inactive. When you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again. For controls—like buttons—that don't retain a setting, this value should be 1. For pop-up menus, the Control Manager sets this value to the number of items in the menu. For other controls, you can use this field as you wish. You can use the `GetControlMaximum` function to determine the value of this field, and you can use the `SetControlMaximum` procedure to change the value of this field.

`contrlDefProc` A handle to the control definition function for this type of control. When you create a control, you identify its type with a control definition ID, which is converted into a handle to the control definition function and stored in this field. Thereafter, the Control Manager uses this handle to access the definition function; you should never need to refer to this field directly.

Note

In systems running in 24-bit mode, the high-order byte of the `contrlDefProc` field contains the variant, which the Control Manager gets from the control definition ID. ◆

`contrlData` Reserved for use by the control definition function, typically to hold additional information specific to a particular control type. For example, the control definition function for scroll bars uses this field for a handle to the region containing the scroll box. (If no more than 4 bytes of additional information are needed, the definition function may store the information directly in the `contrlData` field rather than using a handle.) The control definition function for pop-up menus uses this field to store a pop-up private data record, which is described on page 5-77.

`contrlAction` A pointer to the control's action procedure, if any. The `TrackControl` function may call this procedure to respond to the user's dragging of the control, and this procedure responds by repeatedly performing some action as long as the user holds down the mouse button. See the description of `TrackControl` on page 5-90 for more information about the action procedure. You can use the `GetControlAction` function to determine the current value of this field and the `SetControlAction` procedure to change it.

`contrlRfCon` The control's reference value, which your application may use for any purpose. You can use the `GetControlReference` function to determine the current value of this field and the `SetControlReference` procedure to change it.

`contrlTitle` The control title, if any. You can use the `GetControlTitle` procedure to determine the current value of this field and the `SetControlTitle` procedure to change it.

The Auxiliary Control Record

For drawing all controls on systems running in 32-bit mode (which users can select using the Memory control panel), and for drawing controls that use colors other than the system default, the Control Manager creates and maintains a linked list of auxiliary control records, beginning in the global variable `AuxCtlHead`. (There is only one global list for all controls in all windows, not a separate one for each window. Each window record, by contrast, has a handle to the list of its own controls.)

An auxiliary control record is a data structure of type `AuxCtlRec`. Your application doesn't create and generally shouldn't manipulate an auxiliary control record for a control; rather, you let the Control Manager create and manipulate the auxiliary control record. To create controls using colors other than the system default colors, use the `SetControlColor` procedure (described on page 5-101) or create a control color table resource (described on page 5-121) and let the Control Manager create the necessary auxiliary control records. There is, however, a field in the auxiliary control record that you can use to store information as you see fit; to get a handle to the auxiliary control record for a control, you can use the `GetAuxiliaryControlRecord` function (described on page 5-107).

Each auxiliary control record is relocatable and resides in your application heap. Here is how an auxiliary control record is defined:

```
TYPE AuxCtlRec =
RECORD
   acNext:        AuxCtlHandle;  {handle to next AuxCtlRec}
   acOwner:       ControlHandle; {handle to this record's control}
   acCTable:      CCTabHandle;   {handle to control color table }
                                 { record}
   acFlags:       Integer;       {reserved}
   acReserved:    LongInt;       {reserved for future use}
   acRefCon:      LongInt;       {for use by application}
END;
```

Field descriptions

`acNext`	A handle to the next record in the auxiliary control list.
`acOwner`	The handle of the control to which this auxiliary record belongs; used as an ID field.
`acCTable`	The handle to a control color table record. (The control color table record is described on page 5-77.)
`acFlags`	Reserved for use by the Control Manager.
`acReserved`	Reserved for future expansion.
`acRefCon`	A reference value, which your application may use for any purpose.

On systems using 32-bit mode, every control has its own auxiliary record, and the `acCTable` field contains a handle to the default control color table unless your application uses the `SetControlColor` procedure or creates a control color table resource.

When drawing a control, the standard control definition functions search the linked list of auxiliary control records for the auxiliary control record whose `acOwner` field points to the control being drawn. If the standard control definition functions find an auxiliary control record for the control, they use the control color table specified in the `acCTable` field. If the standard control definition functions do not find an auxiliary control record for the control, they use the default system colors.

The Pop-Up Menu Private Data Record

You can obtain the menu handle and the menu ID of the menu associated with a pop-up menu by dereferencing the `contrlData` field of the pop-up menu's control record. The `contrlData` field of a control record is a handle to a block of private information. For pop-up menu controls, this field is a handle to a pop-up private data record, which is a data structure of type `popupPrivateData`.

```
TYPE  popupPrivateData =
      RECORD
         mHandle:    MenuHandle;       {handle to menu record}
         mID:        Integer;          {menu ID}
         mPrivate:   ARRAY[0..0] OF SignedByte; {reserved}
      END;
```

Field descriptions

`mHandle`	Contains a handle to the menu.
`mID`	The menu ID of the menu.
`mPrivate`	Reserved.

You can use the standard pop-up control definition function to manage pop-up menus. For information on creating pop-up menus, see "Creating a Pop-Up Menu" beginning on page 5-25. See the chapter "Menu Manager" in this book for additional information.

The Control Color Table Record

By creating a control color table record and using the `SetControlColor` procedure (described on page 5-101), your application can draw a control that uses colors other than the system default. (Alternatively, you can use nonstandard colors for a control you define in a control resource by creating a control color table resource—described on page 5-121—with the same resource ID as the control resource.) Be aware that controls in nonstandard colors may initially confuse your users.

A control color table record is a data structure of type `CtlCTab`; it is defined as follows:

```
TYPE CtlCTab =
RECORD
   ccSeed:     LongInt;    {reserved; set to 0}
   ccRider:    Integer;    {reserved; set to 0}
```

```
   ctSize:     Integer;    {number of ColorSpec records in next }
                           { field; 3 for standard controls}
   ctTable:    ARRAY[0..3] OF ColorSpec;
END;
```

Field descriptions

`ccSeed` Reserved in control color tables; set to 0.

`ccRider` Reserved in control color tables; set to 0.

`ctSize` The number of `ColorSpec` records in the next field. For controls drawn with the standard definition procedure, this field is always 3, because a standard control has three parts: frame, control body, and scroll box for scroll bars, and frame, control body, and text for other controls. If you want to supply `ColorSpec` records for additional parts, you must define your own controls, as described in "Defining Your Own Control Definition Function" beginning on page 5-109.

`ctTable` An array of `ColorSpec` records. Each `ColorSpec` record describes the color of a different control part. Here is how a `ColorSpec` record is defined:

```
TYPE ColorSpec =
RECORD
   partIdentifier:   Integer;    {control part}
   partRGB:          RGBColor;   {color of part}
END;
```

The `partIdentifier` field of the `ColorSpec` record holds an integer that associates an `RGBColor` record with a particular part of the control.

Three `ColorSpec` records are used to describe the parts of buttons, checkboxes, and radio buttons. Here are the constants that are used in the `partIdentifier` fields of the three `ColorSpec` records used to describe these controls:

```
{for buttons, checkboxes, and radio buttons}
CONST cFrameColor = 0;  {frame color}
      cBodyColor = 1;   {fill color for body of }
                        { control}
      cTextColor = 2;   {text color}
```

When highlighted, buttons exchange their body and text colors; checkboxes and radio buttons change their appearance without changing colors. All three types indicate deactivation by dimming their text with no change in colors.

A number of `ColorSpec` records are used to describe the parts of scroll bars. Here are the constants that are used in the `partIdentifier` fields of the `ColorSpec` records used to describe the colors in scroll bars:

```
CONST
cFrameColor       = 0;  {Used to produce foreground color for scroll arrows }
                        { & gray area}
cBodyColor        = 1;  {Used to produce colors in the scroll box}
cArrowsColorLight = 5;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cArrowsColorDark  = 6;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cThumbLight       = 7;  {Used to produce colors in scroll box}
cThumbDark        = 8;  {Used to produce colors in scroll box}
cHiliteLight      = 9;  {Use same value as wHiliteColorLight in 'wctb'}
cHiliteDark       = 10; {Use same value as wHiliteColorDark in 'wctb'}
cTitleBarLight    = 11; {Use same value as wTitleBarLight in 'wctb'}
cTitleBarDark     = 12; {Use same value as wTitleBarDark in 'wctb'}
cTingeLight       = 13; {Use same value as wTingeLight in 'wctb'}
cTingeDark        = 14; {Use same value as wTingeDark in 'wctb'}
```

When highlighted, scroll arrows are filled with the foreground color. A deactivated scroll bar shows no scroll box and displays its gray areas in a solid background color with no pattern.

The `ColorSpec` records for a control can appear in any order. If you include a part identifier that is not found, the Control Manager uses the first `ColorSpec` record with an identifiable part. If you do not specify a part identifier, the Control Manager uses the default color for that part.

The `partRGB` field of the `ColorSpec` record specifies an `RGBColor` record, which in turn specifies the red, green, and blue values for the part's color. Use three 16-bit unsigned integers to give the intensity values for the three additive primary colors. Here is how the `RGBColor` record is defined:

```
TYPE RGBColor =
RECORD
   red:     Integer; {red value for control part}
   green:   Integer; {green value for control part}
   blue:    Integer; {blue value for control part}
END;
```

When you create a control color table record, your application should not deallocate it if another control is still using it.

When drawing a control, the standard control definition functions search the linked list of auxiliary control records for the record whose `acOwner` field points to that control. If a standard control definition function finds such a record, it uses the color table designated by that record; otherwise, it uses the default system colors. Each control

using colors other than the system default has its own auxiliary control record, even if that control uses the same control color table record as another control; two or more auxiliary records can share the same control color table record. (Auxiliary control records are described on page 5-76.)

If you create a control definition function (as explained in "Defining Your Own Control Definition Function" beginning page 5-109), you can use color tables of any desired size and define their contents in any way you wish, except that part indices 1 through 127 are reserved for system definition. Any such nonstandard control definition function should bypass the defaulting mechanism by allocating an explicit auxiliary record for every control it creates.

Control Manager Routines

This section describes the Control Manager routines for creating controls, drawing controls, tracking mouse events within controls, changing control display, determining control values, and removing controls.

Some Control Manager routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, `SetControlValue` is also available as `SetCtlValue`. Table 5-1 provides a mapping between the previous name of a routine and its new equivalent name.

Table 5-1 Mapping between new and previous names of Control Manager routines

New name	Previous name
`GetAuxiliaryControlRecord`	`GetAuxCtl`
`GetControlAction`	`GetCtlAction`
`GetControlMaximum`	`GetCtlMax`
`GetControlMinimum`	`GetCtlMin`
`GetControlReference`	`GetCRefCon`
`GetControlTitle`	`GetCTitle`
`GetControlValue`	`GetCtlValue`
`GetControlVariant`	`GetCVariant`
`SetControlAction`	`SetCtlAction`
`SetControlColor`	`SetCtlColor`
`SetControlMaximum`	`SetCtlMax`
`SetControlMinimum`	`SetCtlMin`
`SetControlReference`	`SetCRefCon`
`SetControlTitle`	`SetCTitle`
`UpdateControls`	`UpdtControl`

Creating Controls

To create a control, you should generally use the `GetNewControl` function, which takes information about the control from a control resource. Like menu resources, control resources isolate descriptive information from your application code, making your application easier to modify or translate. However, you can also use the `NewControl` function—for which you pass descriptive information in parameters—to create controls.

Both `GetNewControl` and `NewControl` return a handle to the control record of the newly created control. Thereafter, your application normally refers to the control by this handle, because most other Control Manager routines expect a control handle as their first parameter. When you create scroll bars and pop-up menus, you should store their handles in one of your application's own data structures for later reference.

When you use the Dialog Manager to implement buttons, radio buttons, checkboxes, and pop-up menus in alert boxes and dialog boxes, the Dialog Manager automatically uses the Control Manager to create these controls for you. If you implement other controls in alert or dialog boxes, and whenever you implement controls—such as scroll bars—in your application's windows, you must use either `GetNewControl` or `NewControl` to create these controls.

GetNewControl

To create a control from a description in a control resource (`'CNTL'`), use the `GetNewControl` function.

```
FUNCTION GetNewControl (controlID: Integer; owner: WindowPtr)
                        : ControlHandle;
```

`controlID` The resource ID of a control resource.

`owner` A pointer to the window in which you want to attach the control.

DESCRIPTION

The `GetNewControl` function creates a control record from the information in the specified control resource, adds the control record to the control list for the specified window, and returns as its function result a handle to the control. You use this handle when referring to the control in most other Control Manager routines. After making a copy of the control resource, `GetNewControl` releases the memory occupied by the original control resource before returning.

If you provide a control color table resource with the same resource ID as the control resource, `GetNewControl` creates an auxiliary control record that uses the colors you specify in your control color table resource. If you don't provide a control color table, `GetNewControl` creates an auxiliary control record that uses the default control color table if the computer is running in 32-bit mode.

The control resource specifies the rectangle for the control, its initial setting, its visibility state, its maximum and minimum settings, its control definition ID, a reference value, and its title (if any). After you use `GetNewControl` to create the control, you can change the current setting, the maximum setting, the minimum setting, the reference value, and the title by using, respectively, the `SetControlValue`, `SetControlMaximum`, `SetControlMinimum`, `SetControlReference`, and `SetControlTitle` procedures. You can use the `MoveControl` and `SizeControl` procedures to change the control's rectangle. You can use the `GetControlValue`, `GetControlMaximum`, `GetControlMinimum`, `GetControlReference`, and `GetControlTitle` functions to determine the control values.

If the control resource specifies that the control should be visible, the Control Manager draws the control. If the control resource specifies that the control should initially be invisible, you can use the `ShowControl` procedure to make the control visible.

If `GetNewControl` can't read the control resource from the resource file, `GetNewControl` returns `NIL`.

SEE ALSO

See Listing 5-1 on page 5-17 and Listing 5-5 on page 5-24 for examples of how to use `GetNewControl` to create, respectively, a button and a scroll bar. For information about windows' control lists, see the chapter "Window Manager" in this book.

NewControl

To create a control, you can use the `NewControl` function, which accepts in its parameters the information that describes the control. Generally, you should instead use the `GetNewControl` function to create a control. The `GetNewControl` function takes information about the control from a control resource, and as a result your application is easier to modify or translate into other languages.

```
FUNCTION NewControl (theWindow: WindowPtr; boundsRect: Rect;
                     title: Str255; visible: Boolean;
                     value: Integer; min: Integer; max: Integer;
                     procID: Integer; refCon: LongInt)
                     : ControlHandle;
```

`theWindow` A pointer to the window in which you want to attach the control. All coordinates pertaining to the control are interpreted in this window's local coordinate system.

`boundsRect` The rectangle, specified in the given window's local coordinates, that encloses the control and thus determines its size and location.

`title` For controls that need a title—such as buttons, checkboxes, radio buttons, and pop-up menus—the string for that title. For controls that don't use titles, pass an empty string.

`visible` The visible/invisible state for the control. If you pass `TRUE` in this parameter, `NewControl` draws the control immediately, without using your window's standard updating mechanism. If you pass `FALSE`, you must later use the `ShowControl` procedure to display the control.

`value` The initial setting for the control. For controls—such as buttons—that don't retain a setting, pass 0 in this parameter. For controls—such as checkboxes and radio buttons—that retain an on-or-off setting, pass 0 in this parameter for a control that is off, and pass 1 for a control that is on. For controls—such as scroll bars and sliders—that can take a range of settings, specify whatever value is appropriate within that range.

`min` The minimum setting for the control. For controls—such as buttons—that don't retain a setting, pass 0 in this parameter. For controls—such as checkboxes and radio buttons—that retain an on-or-off setting, use 0 (meaning "off") for the minimum value. For controls—such as scroll bars and sliders—that can take a range of settings, specify whatever minimum value is appropriate.

`max` The maximum setting for the control. For controls—such as buttons—that don't retain a setting, pass 1 in this parameter. For controls—such as checkboxes and radio buttons—that retain an on-or-off setting, use 1 (meaning "on") for the maximum value. For controls—such as scroll bars and sliders—that can take a range of settings, specify whatever maximum value is appropriate. When you make the maximum setting of a scroll bar equal to its minimum setting, the control definition function automatically makes the scroll bar inactive; when you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.

`procID` The control definition ID, which leads to the control definition function for this type of control. The control definition function is read into memory if it isn't already in memory. The control definition IDs and their constants for the standard controls are listed here. (You can also define your own control definition function and specify it the `procID` parameter.)

```
CONST
   pushButProc        = 0;     {button}
   checkBoxProc       = 1;     {checkbox}
   radioButProc       = 2;     {radio button}
   useWFont           = 8;     {add to above to display }
                               { title in window font}
   scrollBarProc      = 16;    {scroll bar}
   popupMenuProc      = 1008;  {pop-up menu}
   popupFixedWidth    = $0001; {add to popupMenuProc to }
                               { use a fixed-width ctrl}
```

```
popupUseAddResMenu = $0004; {add to popupMenuProc to }
                            { specify a value of }
                            { type ResType in the }
                            { contrlRfCon field of }
                            { the control record; }
                            { Menu Manager adds }
                            { resources of this type }
                            { to the menu}
popupUseWFont      = $0008; {add to popupMenuProc to }
                            { display in window font}
```

`refCon` The control's reference value, which is set and used only by your application.

DESCRIPTION

The `NewControl` function creates a control record from the information you specify in its parameters, adds the control record to the control list for the specified window, and returns as its function result a handle to the control. You use this handle when referring to the control in most other Control Manager routines.

The `NewControl` function creates an auxiliary control record that uses the default control color table if the computer is running in 32-bit mode.

If you need to use colors other than the default colors for the control, create a control color table record and use the `SetControlColor` procedure.

When specifying the rectangle in the `boundsRect` parameter, keep the following guidelines in mind:

- Buttons are drawn to fit the rectangle exactly. To accommodate the tallest characters in the system font, allow at least a 20-point difference between the top and bottom coordinates of the rectangle.
- For checkboxes and radio buttons, there should be at least a 16-point difference between the top and bottom coordinates.
- By convention, scroll bars are 16 pixels wide, so there should be a 16-point difference between the left and right (or top and bottom) coordinates. (If there isn't, the scroll bar is scaled to fit the rectangle.) A standard scroll bar should be at least 48 pixels long, to allow room for the scroll arrows and scroll box.

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

The Control Manager allows multiple lines of text in the titles of buttons, checkboxes, and radio buttons. When specifying a multiple-line title, separate the lines with the ASCII character code $0D (carriage return). If the control is a button, each line is horizontally centered, and the font leading is inserted between lines. (The height of each

line is equal to the distance from the ascent line to the descent line plus the leading of the font used. Be sure to make the total height of the rectangle greater than the number of lines times this height.) If the control is a checkbox or a radio button, the text is justified as appropriate for the user's current script system, and the checkbox or button is vertically centered within its rectangle.

After you use `NewControl` to create the control, you can change the current setting, the maximum setting, the minimum setting, the reference value, and the title by using, respectively, the `SetControlValue`, `SetControlMaximum`, `SetControlMinimum`, `SetControlReference`, and `SetControlTitle` procedures. You can use the `MoveControl` and `SizeControl` procedures to change the control's rectangle. You can use the `GetControlValue`, `GetControlMaximum`, `GetControlMinimum`, `GetControlReference`, and `GetControlTitle` functions to determine the control values.

SPECIAL CONSIDERATIONS

The title of a button, checkbox, radio button, or pop-up menu normally appears in the system font, which in Roman script systems is 12-point Chicago. Do not use a smaller font; some script systems, such as KanjiTalk, require 12-point fonts. You should generally use the system font in your controls; doing so will simplify localization effort. However, if you absolutely need to display a control title in the font currently associated with the window's graphics port, you can add the `popupUseWFont` constant to the pop-up menu control definition ID or add the `useWFont` constant to the other standard control definition IDs.

SEE ALSO

For information about windows' control lists, see the chapter "Window Manager" in this book. Control definition IDs for other controls are discussed in "Defining Your Own Control Definition Function" beginning on page 5-109.

Drawing Controls

If you specify that a control is initially visible (either in the control resource or in a parameter to `NewControl`), the Control Manager draws the control inside its window when you call either the `GetNewControl` or the `NewControl` function. In either case, the Control Manager draws the control immediately, without using your window's standard updating mechanism. If you specify that a control is invisible, you can use the `ShowControl` procedure when you want to draw the control.

Note that even a visible control might be completely or partially obscured by overlapping windows or other objects.

When your application receives an update event for a window that contains controls, use `UpdateControls` to redraw the necessary controls in the updated window. Note that the Dialog Manager automatically draws and updates controls in alert boxes and dialog boxes.

ShowControl

To draw a control that is currently invisible, you can use the `ShowControl` procedure.

```
PROCEDURE ShowControl (theControl: ControlHandle);
```

`theControl` A handle to the control you want to make visible.

DESCRIPTION

If the specified control is invisible, the `ShowControl` procedure makes it visible and immediately draws the control within its window without using your window's standard updating mechanism. If the control is already visible, `ShowControl` has no effect.

You can make a control invisible in several ways:

- You can specify that it's invisible in its control resource.
- You can specify that it's invisible in a parameter to the `NewControl` function.
- You can use the `HideControl` procedure to change a visible control into an invisible one.
- You can directly change the `contrlVis` field of the control's control record.

SPECIAL CONSIDERATIONS

The `ShowControl` procedure draws the control in its window, but the control can still be completely or partially obscured by overlapping windows or other objects.

SEE ASO

Listing 5-14 on page 5-39 illustrates the use of `ShowControl` to redisplay scroll bars after moving and resizing them.

UpdateControls

To update controls in a window, you can use the `UpdateControls` procedure. The `UpdateControls` procedure is also available as the `UpdtControl` procedure.

```
PROCEDURE UpdateControls (theWindow: WindowPtr;
                          updateRgn: RgnHandle);
```

`theWindow` A pointer to the window containing the controls to update.

`updateRgn` The update region within the specified window.

DESCRIPTION

The `UpdateControls` procedure draws those controls that are in the specified update region. This procedure is faster than the `DrawControls` procedure, which draws *all* of the controls in a window. By contrast, `UpdateControls` draws only those controls in the update region.

Your application should call `UpdateControls` upon receiving an update event for a window that contains controls. Window Manager routines such as `SelectWindow`, `ShowWindow`, and `BringToFront` do not automatically call `DrawControls` to display the window's controls. They just add the appropriate regions to the window's update region, generating an update event.

In response to an update event, you normally call `UpdateControls` after using the Window Manager procedure `BeginUpdate` and before using the Window Manager procedure `EndUpdate`. You should set the `updateRgn` parameter to the visible region of the window's port, as specified in the port's `visRgn` field.

SPECIAL CONSIDERATIONS

If your application draws parts of a control outside of its rectangle, `UpdateControls` might not redraw it.

The Dialog Manager handles update events for controls in alert boxes and dialog boxes.

SEE ALSO

Listing 5-8 on page 5-30 illustrates the use of `UpdateControls`. The `BeginUpdate` and `EndUpdate` procedures are described in the chapter "Window Manager" in this book. See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes.

DrawControls

Although you should generally use the `UpdateControls` procedure to update controls in a window, you can instead use the `DrawControls` procedure.

```
PROCEDURE DrawControls (theWindow: WindowPtr);
```

`theWindow` A pointer to a window whose controls you want to display.

DESCRIPTION

The `DrawControls` procedure draws *all* controls currently visible in the specified window. The controls are drawn in reverse order of creation; thus, in case of overlapping controls, the control created first appears frontmost in the window.

Because the `UpdateControls` procedure redraws only those controls that need updating, your application should generally use it instead of `DrawControls` upon receiving an update event for a window that contains controls.

You should call either `DrawControls` or `UpdateControls` after calling the Window Manager procedure `BeginUpdate` and before calling `EndUpdate`.

SPECIAL CONSIDERATIONS

The Dialog Manager automatically draws and updates controls in alert boxes and dialog boxes.

Window Manager routines such as `SelectWindow`, `ShowWindow`, and `BringToFront` do not automatically update the window's controls. They just add the appropriate regions to the window's update region, generating an update event.

SEE ALSO

See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes. See the chapter "Window Manager" in this book for more information about Window Manager routines.

Draw1Control

Although you should generally use the `UpdateControls` procedure to update controls, you can use the `Draw1Control` procedure to update a single control.

```
PROCEDURE Draw1Control (theControl: ControlHandle);
```

`theControl` A handle to the control you want to draw.

DESCRIPTION

The `Draw1Control` procedure draws the specified control if it's visible within its window. The `UpdateControls` procedure automatically calls `Draw1Control`.

Handling Mouse Events in Controls

When the user presses the mouse button, your application receives a mouse-down event. Use the Window Manager function `FindWindow` to determine which window contains the cursor. If the mouse-down event occurred in the content region of your application's active window, use the `FindControl` function to determine whether the cursor was in an active control and, if so, which control. To follow and respond to the cursor movements in that control, and then to determine in which part of the control the mouse-up event occurs, use the `TrackControl` function.

FindControl

To determine whether a mouse-down event has occurred in a control and, if so, in which part of that control, use the `FindControl` function.

```
FUNCTION FindControl (thePoint: Point; theWindow: WindowPtr;
                      VAR theControl: ControlHandle): Integer;
```

`thePoint` A point, specified in coordinates local to the window, where the mouse-down event occurred.

`theWindow` A pointer to the window in which the mouse-down event occurred.

`theControl` A handle to the control in which the mouse-down event occurred.

DESCRIPTION

When the user presses the mouse button while the cursor is in a visible, active control, `FindControl` returns as its function result a part code identifying the control's part; the function also returns a handle to the control in the parameter `theControl`. The part codes that `FindControl` returns, and the constants you can use to represent them, are listed here:

```
CONST inButton       = 10;    {button}
      inCheckBox     = 11;    {checkbox or radio button}
      inUpButton     = 20;    {up arrow for a vertical scroll }
                              { bar, left arrow for a horizontal }
                              { scroll bar}
      inDownButton   = 21;    {down arrow for a vertical scroll }
                              { bar, right arrow for a }
                              { horizontal scroll bar}
      inPageUp       = 22;    {gray area above scroll box for a }
                              { vertical scroll bar, gray area }
                              { to left of scroll box for a }
                              { horizontal scroll bar}
      inPageDown     = 23;    {gray area below scroll box for a }
                              { vertical scroll bar, gray area }
                              { to right of scroll box for a }
                              { horizontal scroll bar}
      inThumb        = 129;   {scroll box}
```

The pop-up control definition function does not define part codes for pop-up menus. Instead, your application should store the handles for your pop-up menus when you create them. Your application should then test the handles you store against the handles returned by `FindControl` before responding to users' choices in pop-up menus.

If the mouse-down event occurs in an invisible or inactive control, or if it occurs outside a control, `FindControl` sets `theControl` to `NIL` and returns 0 as its function result.

When a mouse-down event occurs, your application should call `FindControl` after using the Window Manager function `FindWindow` to ascertain that a mouse-down event has occurred in the content region of a window containing controls.

Before calling `FindControl`, use the `GlobalToLocal` procedure to convert the point stored in the `where` field (which describes the location of the mouse-down event) of the event record to coordinates local to the window. Then, when using `FindControl`, pass this point in the parameter `thePoint`.

In the parameter `theWindow`, pass the window pointer returned by the `FindWindow` function.

After using `FindControl` to determine that a mouse-down event has occurred in a control, you generally use the `TrackControl` function, which automatically follows the movements of the cursor and responds as appropriate until the user releases the mouse button.

SPECIAL CONSIDERATIONS

The Dialog Manager automatically calls `FindControl` and `TrackControl` for mouse-down events inside controls of alert boxes and dialog boxes.

The `FindControl` function also returns `NIL` in the parameter `theControl` and 0 as its function result if the window is invisible or if it doesn't contain the given point. (However, `FindWindow` won't return a window pointer to an invisible window or to one that doesn't contain the point where the mouse-down event occurred. As long as you call `FindWindow` before `FindControl`, this situation won't arise.)

SEE ALSO

Listing 5-10 on page 5-33 illustrates the use of `FindControl` for detecting mouse-down events in a pop-up menu and a button; Listing 5-18 on page 5-53 illustrates its use for detecting mouse-down events in scroll bars.

The `FindWindow` function is described in the chapter "Window Manager" in this book. The `GlobalToLocal` procedure is described in *Inside Macintosh: Imaging*.

The event record is described in the chapter "Event Manager" in this book. See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes.

TrackControl

To follow and respond to cursor movements in a control and then to determine the control part in which the mouse-up event occurs, use the `TrackControl` function.

```
FUNCTION TrackControl (theControl: ControlHandle;
                       thePoint: Point; actionProc: ProcPtr)
                       : Integer;
```

`theControl` A handle to the control in which a mouse-down event occurred.

`thePoint` A point, specified in coordinates local to the window, where the mouse-down event occurred.

`actionProc` The action procedure. Typically, you should set this parameter to `NIL` for buttons, checkboxes, radio buttons, and the scroll box of a scroll bar; set this parameter to `Pointer(-1)` for pop-up menus; and set this parameter to the pointer to an action procedure for scroll arrows and gray areas of scroll bars, as well as for any other controls that require you to define additional actions to take while the user holds down the mouse button.

DESCRIPTION

The `TrackControl` function follows the user's cursor movements in a control and provides visual feedback until the user releases the mouse button. The visual feedback given by `TrackControl` depends on the control part in which the mouse-down event occurs. When highlighting is appropriate, for example, `TrackControl` highlights the control part (and removes the highlighting when the user releases the mouse button). When the user holds down the mouse button while the cursor is in an indicator (such as the scroll box of a scroll bar) and moves the mouse, `TrackControl` responds by dragging a dotted outline of the indicator.

The `TrackControl` function returns as its function result the control's part code if the user releases the mouse button while the cursor is inside the control part, or 0 if the user releases the mouse button while the cursor is outside the control part. For control parts, the `TrackControl` function returns the same values (represented by the constants `inButton`, `inCheckBox`, `inUpButton`, `inDownButton`, `inPageUp`, `inPageDown`, and `inThumb`) returned by the `FindControl` function, as described on page 5-89.

When `TrackControl` returns a value other than 0 as its function result, your application should respond as appropriate to a mouse-up event in that control part. When `TrackControl` returns 0 as its function result, your application should do nothing.

If the user releases the mouse button when the cursor is in an indicator such as a scroll box, `TrackControl` calls the control's control definition function to reposition the indicator. The control definition function for scroll bars, for example, responds to the user dragging a scroll box by redrawing the scroll box, calculating the control's current setting according to the new relative position of the scroll box, and storing the current setting in the control record. Thus, if the minimum and maximum settings are 0 and 10, and the scroll box is in the middle of the scroll bar, 5 is stored as the current setting. For a scroll bar, your application must then respond by scrolling to the corresponding relative position in the document.

Generally, you use `TrackControl` after using the `FindControl` function. In the parameter `theControl` of `TrackControl`, pass the control handle returned by the `FindControl` function, and in the parameter `thePoint`, supply the same point you passed to `FindControl` (that is, a point in coordinates local to the window).

While the user holds down the mouse button with the cursor in one of the standard controls, `TrackControl` performs the following actions, depending on the value you pass in the parameter `actionProc`. (For other controls, what you pass in this parameter depends on how you define the control.)

- If you pass `NIL` in the `actionProc` parameter, `TrackControl` uses no action procedure and therefore performs no additional actions beyond highlighting the control or dragging the indicator. This is appropriate for buttons, checkboxes, radio buttons, and the scroll box of a scroll bar.
- If you pass a pointer to an action procedure in the `actionProc` parameter, you must provide the procedure, and it must define some action that your application repeats as long as the user holds down the mouse button. This is appropriate for the scroll arrows and gray areas of a scroll bar.
- If you pass `Pointer(–1)` in the `actionProc` parameter, `TrackControl` looks in the `contrlAction` field of the control record for a pointer to the control's action procedure. This is appropriate when you are tracking the cursor in a pop-up menu. (You can use the `GetControlAction` function to determine the value of this field, and you can use the `SetControlAction` procedure to change this value.) If the `contrlAction` field of the control record contains a procedure pointer, `TrackControl` uses the action procedure it points to; if the field of the control record also contains the value `Pointer(–1)`, `TrackControl` calls the control's control definition function to perform the necessary action; you may wish to do this if you define your own control definition function for a custom control. If the field of the control record contains the value `NIL`, `TrackControl` performs no action.

SPECIAL CONSIDERATIONS

When you need to handle events in alert and dialog boxes, Dialog Manager routines automatically call `FindControl` and `TrackControl`.

ASSEMBLY-LANGUAGE INFORMATION

The `TrackControl` function invokes the Window Manager function `DragGrayRgn`, so you can use the global variables `DragHook` and `DragPattern`.

SEE ALSO

See "Defining Your Own Action Procedures" beginning on page 5-115 for information about an action procedure to specify in the `actionProc` parameter. See "Defining Your Own Control Definition Function" beginning on page 5-109 for information about creating a control definition function.

Listing 5-11 on page 5-36, Listing 5-12 on page 5-37, Listing 5-13 on page 5-38, and Listing 5-18 on page 5-53 illustrate the use of `TrackControl` for responding to mouse-down events in, respectively, a button, a pop-up menu, a checkbox, and a scroll bar.

See the chapter "Dialog Manager" in this book for more information about including controls in alert boxes and dialog boxes.

TestControl

The `TestControl` function is called by the `FindControl` and `TrackControl` functions—normally you won't need to call it yourself. However, should you ever need to determine the control part in which a mouse-down event occurred, you can use the `TestControl` function.

```
FUNCTION TestControl (theControl: ControlHandle; thePt: Point)
                      : Integer;
```

`theControl` A handle to the control in which the mouse-down event occurred.

`thePt` The point, in a window's local coordinates, where the mouse-down event occurred.

DESCRIPTION

When the control specified by the parameter `theControl` is visible and active, `TestControl` tests which part of the control contains the point specified by the parameter `thePt`. For its function result, `TestControl` returns the part code of the control part, or 0 if the point is outside the control.

If the control is invisible or inactive, `TestControl` returns 0.

Changing Control Settings and Display

In response to user actions, you often need to change the settings, highlight states, sizes, and locations of your controls. Whenever your application calls the `TrackControl` function, the Control Manager automatically manipulates control display as appropriate as the user presses and releases the mouse button. For example, `TrackControl` calls the `HiliteControl` procedure to highlight buttons; for scroll bars, `TrackControl` calls the `DragControl` procedure to move an outline of the scroll box in a scroll bar and the `SetControlValue` procedure to change the scroll bar's current setting and redraw the scroll box in its new location. (Note that the Dialog Manager automatically calls `TrackControl` for controls in alert boxes and dialog boxes. See the chapter "Dialog Manager" in this book for more information.)

When the user releases the mouse button while the cursor is in a control, your application often needs to change its setting. When the user clicks a checkbox, for example, your application must change its setting to on or off, and the Control Manager automatically draws or removes an X in the checkbox.

There are other instances when you must change the settings and display of a control. For example, when the user changes the size of a window that contains a scroll bar, you need to resize and move the scroll bar accordingly.

For controls whose values the user can set, you can use the `SetControlValue` procedure to change the control's setting and redraw the control accordingly. When you need to change the maximum setting of a scroll bar or a dial, you can use the

SetControlMaximum procedure; if you need to change the minimum setting, you can use the SetControlMinimum procedure. If you need to change a control title, you can use the SetControlTitle procedure. You can use the HideControl procedure to make a control invisible. When you need to make a control inactive (such as when its window is not frontmost) or in any other way change the highlighting of a control, you can use the HiliteControl procedure.

To move a scroll bar, you use the MoveControl and SizeControl procedures.

Although it's not recommended, you can also change a control's default colors to those of your own choosing by using the SetControlColor procedure.

To invoke a continuous action while the user holds down the mouse button, you can specify an action procedure (described in "Defining Your Own Action Procedures" beginning on page 5-115) in a parameter to TrackControl. Under certain circumstances, you can use the SetControlAction procedure to change the control's action procedure, though you should rarely if ever need to.

SetControlValue

To change the current setting of a control and redraw it accordingly, you can use the SetControlValue procedure. The SetControlValue procedure is also available as the SetCtlValue procedure.

```
PROCEDURE SetControlValue (theControl: ControlHandle;
                           theValue: Integer);
```

theControl A handle to the control whose current setting you wish to change.

theValue The new setting for the control.

DESCRIPTION

The SetControlValue procedure changes the contrlValue field of the control record to the specified value and redraws the control to reflect the new setting. For checkboxes and radio buttons, the value 1 fills the control with the appropriate mark, and 0 removes the mark. For scroll bars, SetControlValue redraws the scroll box where appropriate.

If the specified value is less than the minimum setting for the control, SetControlValue sets the control to its minimum setting; if the value is greater than the maximum setting, SetControlValue sets the control to its maximum.

When you create a control, you specify an initial setting either in the control resource or in the value parameter of the NewControl function. To determine a control's current setting before changing it in response to a user's click in that control, use the GetControlValue function.

SEE ALSO

Listing 5-13 on page 5-38 illustrates the use of `SetControlValue` to change the setting of a checkbox. Listing 5-16 on page 5-41 and Listing 5-20 on page 5-61 illustrate the use of `SetControlValue` to change the setting of a scroll bar.

SetControlMinimum

To change the minimum setting of a control and redraw its indicator or scroll box accordingly, you can use the `SetControlMinimum` procedure. The `SetControlMinimum` procedure is also available as the `SetCtlMin` procedure.

```
PROCEDURE SetControlMinimum (theControl: ControlHandle;
                             minValue: Integer);
```

`theControl` A handle to the control whose minimum setting you wish to change.

`minValue` The new minimum setting.

DESCRIPTION

The `SetControlMinimum` procedure changes the `contrlMin` field of the control record to the setting you specify in the `minValue` parameter and redraws its indicator or scroll box to reflect its new range.

When you create a control, you specify an initial minimum setting either in the control resource or in the `min` parameter of the `NewControl` function. To determine a control's current minimum setting, use the `GetControlMinimum` function.

SetControlMaximum

To change the maximum setting of a control and redraw its indicator or scroll box accordingly, you can use the `SetControlMaximum` procedure. The `SetControlMaximum` procedure is also available as the `SetCtlMax` procedure.

```
PROCEDURE SetControlMaximum (theControl: ControlHandle;
                             maxValue: Integer);
```

`theControl` A handle to the control whose maximum setting you wish to change.

`maxValue` The new maximum setting.

DESCRIPTION

The `SetControlMaximum` procedure changes the `contrlMax` field of the control record to the setting you specify in the `maxValue` parameter and redraws its indicator or scroll box to reflect its new range.

When you create a control, you specify an initial maximum setting either in the control resource or in the `max` parameter of the `NewControl` function. To determine a control's current maximum setting, use the `GetControlMaximum` function.

When you set the maximum setting of a scroll bar equal to its minimum setting, the control definition function makes the scroll bar inactive; when you make the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.

SEE ALSO

Listing 5-16 on page 5-41 illustrates the use of `SetControlMaximum` to specify the maximum setting for a scroll bar.

SetControlTitle

To change the title of a control and redraw the control accordingly, use the `SetControlTitle` procedure. The `SetControlTitle` procedure is also available as the `SetCTitle` procedure.

```
PROCEDURE SetControlTitle (theControl: ControlHandle;
                           title: Str255);
```

`theControl` A handle to a control, the title of which you want to change.

`title` The new title for the control.

DESCRIPTION

The `SetControlTitle` procedure changes the `contrlTitle` field of the control record to the given string and redraws the control, using the system font for the control title.

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

When you create a control, you specify an initial title either in the control resource or in the `title` parameter of the `NewControl` function. To determine a control's current title, use the `GetControlTitle` procedure.

HideControl

To make a control invisible, before adjusting its size and location, for example, use the `HideControl` procedure.

```
PROCEDURE HideControl (theControl: ControlHandle);
```

`theControl` A handle to the control you want to hide.

DESCRIPTION

The `HideControl` procedure makes the specified control invisible by changing the value of the `contrlVis` field of the control record and removing the control from the screen. To fill the region previously occupied by the control, `HideControl` uses the background pattern of the window's graphics port. It also adds the control's rectangle to the window's update region, so that anything else that was previously obscured by the control will reappear on the screen. If the control is already invisible, `HideControl` has no effect.

To make the control visible again, you can use the `ShowControl` procedure.

SPECIAL CONSIDERATIONS

The `MoveControl` and `SizeControl` procedures both call `HideControl` and `ShowControl` automatically. However, so that the control will not blink on the screen when you make both of these calls, you should use `HideControl` to make the control invisible until you are finished manipulating it, and then use `ShowControl`.

SEE ALSO

Listing 5-14 on page 5-39 illustrates the use of `HideControl` before adjusting scroll bar settings and locations.

MoveControl

To move a control within its window, you can use the `MoveControl` procedure.

```
PROCEDURE MoveControl (theControl: ControlHandle;
                       h: Integer; v: Integer);
```

`theControl` A handle to the control you wish to move.

`h` The horizontal coordinate (local to the control's window) of the new location of the upper-left corner of the control's rectangle.

`v` The vertical coordinate (local to the control's window) of the new location of the upper-left corner of the control's rectangle.

DESCRIPTION

The `MoveControl` procedure moves the control to the new location specified by the `h` and `v` parameters, using them to change the rectangle specified in the `contrlRect` field of the control's control record. When the control is visible, `MoveControl` first hides it and then redraws it at its new location.

For example, if the user resizes a document window that contains a scroll bar, your application can use `MoveControl` to move the scroll bar to its new location.

SEE ALSO

Listing 5-24 on page 5-67 illustrates the use of `MoveControl` to change the location of a scroll bar.

SizeControl

To change the size of a control's rectangle, use the `SizeControl` procedure.

```
PROCEDURE SizeControl (theControl: ControlHandle;
                       h: Integer; v: Integer);
```

`theControl` A handle to the control you wish to resize.

`w` The new width, in pixels, of the resized control.

`h` The new height, in pixels, of the resized control.

DESCRIPTION

The `SizeControl` procedure changes the rectangle specified in the `contrlRect` field of the control's control record. The lower-right corner of the rectangle is adjusted so that it has the width and height specified by the `w` and `h` parameters; the position of the upper-left corner is not changed. If the control is currently visible, it's first hidden and then redrawn in its new size. The `SizeControl` procedure uses `HideControl`, which changes the window's update region.

SEE ALSO

Listing 5-24 on page 5-67 illustrates the use of `SizeControl` to change the size of a scroll bar.

HiliteControl

If you need to change the highlighting of a control, you can use the `HiliteControl` procedure.

```
PROCEDURE HiliteControl (theControl: ControlHandle;
                         hiliteState: Integer);
```

`theControl` A handle to the control.

`hiliteState`
A value from 0 through 255 to signify the highlighting of the control. The value of 0 signifies no highlighting for the active control. A value from 1 through 253 signifies a part code designating the part of the (active) control to highlight. (Part codes are explained in the description of `FindControl` on page 5-89.) The value 255 signifies that the control is to be made inactive and drawn accordingly.

DESCRIPTION

The `HiliteControl` procedure calls the control definition function to redraw the control with the highlighting specified in the `hiliteState` parameter. The `HiliteControl` procedure uses the value in this parameter to change the value of the `contrlHilite` field of the control's control record.

Except for scroll bars (which you should hide using the `HideControl` procedure), you should use `HiliteControl` to make all controls inactive when their windows are not frontmost. The `TrackControl` function automatically uses the `HiliteControl` procedure as appropriate; when you use `TrackControl`, you don't need to call `HiliteControl`.

SPECIAL CONSIDERATIONS

The value 254 should not be passed in the `hiliteState` parameter; this value is reserved for future use.

SEE ALSO

The chapter "Dialog Manager" in this book provides several examples of the use of `HiliteControl`.

DragControl

If you need to draw and move an outline of a control or its indicator (such as the scroll box of a scroll bar) while the user drags it, you can use the `DragControl` procedure.

```
PROCEDURE DragControl (theControl: ControlHandle;
                       startPt: Point;
                       limitRect: Rect; slopRect: Rect;
                       axis: Integer);
```

`theControl` A handle to the control to drag.

`startPt` The location of the cursor, expressed in the local coordinates of the control's window, at the time the user first presses the mouse button.

`limitRect` A rectangle—which should normally coincide with or be contained in the window's content region—delimiting the area in which the user can drag the control's outline.

`slopRect` A rectangle that allows some extra space for the user to move the mouse while still constraining the control within the rectangle specified in the `limitRect` parameter.

`axis` The axis along which the user may drag the control's outline. The following list shows the constants you can use—and the values they represent—for constraining the motion along an axis:

```
CONST
   noConstraint = 0; {no constraint}
   hAxisOnly    = 1; {drag along horizontal axis only}
   vAxisOnly    = 2; {drag along vertical axis only}
```

DESCRIPTION

The `DragControl` procedure moves a dotted outline of the control around the screen, following the movements of the cursor until the user releases the mouse button. When the user releases the mouse button, `DragControl` calls `MoveControl`. In turn, `MoveControl` moves the control to the location to which the user dragged it.

The `TrackControl` function automatically uses the `DragControl` procedure as appropriate; when you use `TrackControl`, you don't need to call `DragControl`.

The `startPt`, `limitRect`, `slopRect`, and `axis` parameters have the same meaning as for the Window Manager function `DragGrayRgn`.

SPECIAL CONSIDERATIONS

Before tracking the cursor, `DragControl` calls the control definition function. If you define your own control definition function, you can specify custom dragging behavior.

ASSEMBLY-LANGUAGE INFORMATION

Like `TrackControl`, `DragControl` invokes the Window Manager function `DragGrayRgn`, so you can use the global variables `DragHook` and `DragPattern`.

SEE ALSO

For information about creating your own control definition functions, see "Defining Your Own Control Definition Function" beginning on page 5-109. See the description of the `DragGrayRgn` function in the chapter "Window Manager" in this book for a more complete discussion of the `startPt`, `limitRect`, `slopRect`, and `axis` parameters, which are used identically in the `DragControl` function.

SetControlColor

To draw a control using colors other than the default colors used by system software, you can use the `SetControlColor` procedure. The `SetControlColor` procedure is also available as the `SetCtlColor` procedure.

```
PROCEDURE SetControlColor (theControl: ControlHandle;
                           newColorTable: CCTabHandle);
```

`theControl` A handle to the control whose colors you wish to change.

`newColorTable`
A handle to a control color table record.

DESCRIPTION

The `SetControlColor` procedure changes the color table for the specified control. If the control currently has no auxiliary control record, `SetControlColor` creates one that includes the control color table record specified in the parameter `newColorTable` and adds the auxiliary control record to the head of the auxiliary control list. If there is already an auxiliary record for the control, `SetControlColor` replaces its color table with the contents of the control color table record specified in the parameter `newColorTable`.

To use nonstandard colors for a control, you must create a control color table, either by creating a color control table record and calling `SetControlColor` or by creating a control color table resource. Generally, you use `SetControlColor` when you create a control using `NewControl` and want to use nonstandard colors for it or when you change any control's colors after you've created it. When you want to use nonstandard colors for those controls you create in a control (`'CNTL'`) resource, you should create a control color table (`'cctb'`) resource with the same resource ID as the control resource.

A control whose colors you set with `SetControlColor` should initially be invisible. After using `SetControlColor` to set the control's colors, use the `ShowControl` procedure to make the control visible.

SPECIAL CONSIDERATIONS

On color monitors, the Control Manager automatically draws controls so that they match the colors of the controls used by system software. Be aware that nonstandard colors in your controls may initially confuse your users.

When you create a control color table record, your application should not deallocate it if another control is still using it.

SetControlAction

If you set the action procedure to `Pointer(-1)` when you use `TrackControl`, you can use the `SetControlAction` procedure to set or change the action procedure. The `SetControlAction` procedure is also available as the `SetCtlAction` procedure.

```
PROCEDURE SetControlAction (theControl: ControlHandle;
                            actionProc: ProcPtr);
```

`theControl` A handle to the control whose action procedure you wish to change.

`actionProc` A pointer to an action procedure defining what action your application takes while the user holds down the mouse button.

DESCRIPTION

The `SetControlAction` procedure changes the `contrlAction` field of the control's control record to point to the action procedure specified in the `actionProc` parameter. If the cursor is in the specified control, `TrackControl` calls this action procedure when user holds down the mouse button. You must provide the action procedure, and it must define some action to perform repeatedly as long as the user holds down the mouse button. (The `TrackControl` function always highlights and drags the control as appropriate.)

SPECIAL CONSIDERATIONS

The value in the `contrlAction` field of the control's control record is used by `TrackControl` only if you set the action procedure to `TrackControl` to `Pointer(–1)`.

An action procedure is usually specified in a parameter to `TrackControl`; you generally don't need to call `SetControlAction` to change it.

SEE ALSO

Action procedures are described in "Defining Your Own Action Procedures" beginning on page 5-115.

Determining Control Values

Your application sets a control's various values—such as current setting, minimum and maximum settings, title, reference value, and action procedure—when it creates the control. When the user clicks a control, however, your application often needs to determine the current setting and other possible values of that control. When the user clicks a checkbox, for example, your application must determine whether the box is checked before deciding whether to draw a checkmark inside the checkbox or remove the checkmark.

You can use the `GetControlValue`, `GetControlTitle`, `GetControlMinimum`, `GetControlMaximum`, `GetControlAction`, and `GetControlReference` routines to determine, respectively, a control's current setting, title, minimum setting, maximum setting, action procedure, and reference value. To get a handle to a control's auxiliary control record, you can use the `GetAuxiliaryControlRecord` function; your application can use the `acRefCon` field of an auxiliary control record for any purpose. To determine the variation code that is specified in the control definition function for a particular control, you can use the `GetControlVariant` function. This section also includes a description of the `SetControlReference` procedure, which allows your application to change its reference value for a control.

GetControlValue

To determine a control's current setting, use the `GetControlValue` function. The `GetControlValue` function is also available as the `GetCtlValue` function.

```
FUNCTION GetControlValue (theControl: ControlHandle): Integer;
```

`theControl` A handle to a control.

DESCRIPTION

The `GetControlValue` function returns as its function result the specified control's current setting, which is stored in the `contrlValue` field of the control record.

When you create a control, you specify an initial setting either in the control resource or in the `value` parameter of the `NewControl` function. You can change the setting by using the `SetControlValue` procedure.

SEE ALSO

Listing 5-12 on page 5-37 and Listing 5-13 on page 5-38 illustrate the use of `GetControlValue` for determining the current setting of, respectively, a pop-up menu and a checkbox. Listing 5-16 on page 5-41, Listing 5-18 on page 5-53, and Listing 5-20 on page 5-61 illustrate the use of this function for determining the current setting of a scroll bar.

GetControlMinimum

To determine a control's minimum setting, use the `GetControlMinimum` function. The `GetControlMinimum` function is also available as the `GetCtlMin` function.

```
FUNCTION GetControlMinimum (theControl: ControlHandle): Integer;
```

`theControl` A handle to the control whose minimum value you wish to determine.

DESCRIPTION

The `GetControlMinimum` function returns as its function result the specified control's minimum setting, which is stored in the `contrlMin` field of the control record.

When you create a control, you specify an initial minimum setting either in the control resource or in the `min` parameter of the `NewControl` function. You can change the minimum setting by using the `SetControlMinimum` procedure.

GetControlMaximum

To determine a control's maximum setting, use the `GetControlMaximum` function. The `GetControlMaximum` function is also available as the `GetCtlMax` function.

```
FUNCTION GetControlMaximum (theControl: ControlHandle): Integer;
```

`theControl` A handle to the control whose maximum value you wish to determine.

DESCRIPTION

The `GetControlMaximum` function returns as its function result the specified control's maximum setting, which is stored in the `contrlMax` field of the control record.

When you create a control, you specify an initial maximum setting either in the control resource or in the `max` parameter of the `NewControl` function. You can change the maximum setting by using the `SetControlMaximum` procedure.

SEE ALSO

Listing 5-16 on page 5-41 and Listing 5-20 on page 5-61 illustrate the use of `GetControlMaximum` for determining the maximum scrolling distance of a scroll bar.

GetControlTitle

To determine the title of a control, use the `GetControlTitle` procedure. The `GetControlTitle` procedure is also available as the `GetCTitle` procedure.

```
PROCEDURE GetControlTitle (theControl: ControlHandle;
                           VAR title: Str255);
```

`theControl` A handle to the control whose title you want to determine.

`title` The title of the control.

DESCRIPTION

The `GetControlTitle` procedure returns the specified control title, which is stored in the `contrlTitle` field of the control record.

When you create a control, you specify an initial title either in the control resource or in the `title` parameter of the `NewControl` function. You can change the title by using the `SetControlTitle` procedure.

GetControlReference

To determine a control's current reference value, use the `GetControlReference` function. The `GetControlReference` function is also available as the `GetCRefCon` function.

```
FUNCTION GetControlReference (theControl: ControlHandle): LongInt;
```

`theControl` A handle to the control whose current reference value you wish to determine.

DESCRIPTION

The `GetControlReference` function returns as its function result the current reference value for the specified control.

When you create a control, you specify an initial reference value, either in the control resource or in the `refCon` parameter of the `NewControl` function. The reference value is stored in the `contrlRfCon` field of the control record. You can use this field for any purpose, and you can use the `SetControlReference` procedure, described next, to change this value.

SetControlReference

To change a control's current reference value, use the `SetControlReference` procedure. The `SetControlReference` procedure is also available as the `SetCRefCon` procedure.

```
PROCEDURE SetControlReference (theControl: ControlHandle;
                               data: LongInt);
```

`theControl` A handle to the control whose reference value you wish to change.

`data` The new reference value for the control.

DESCRIPTION

The `SetControlReference` procedure sets the control's reference value to the value you specify in the `data` parameter.

When you create a control, you specify an initial reference value, either in the control resource or in the `refCon` parameter of the `NewControl` function. The reference value is stored in the `contrlRfCon` field of the control record; you can use the `GetControlReference` function to determine the current value. You can use this value for any purpose.

GetControlAction

To get a pointer to the action procedure stored in the `contrlAction` field of the control's control record, use the `GetControlAction` function. The `GetControlAction` function is also available as the `GetCtlAction` function.

```
FUNCTION GetControlAction (theControl: ControlHandle): ProcPtr;
```

`theControl` A handle to a control.

DESCRIPTION

The `GetControlAction` function returns as its function result whatever value is stored in the `contrlAction` field of the control's control record. This field specifies the action procedure that `TrackControl` uses if you set its `actionProc` parameter to `Pointer(-1)`. The action procedure should define an action to take in response to the user's holding down the mouse button while the cursor is in the control. You can use the `SetControlAction` procedure to change this action procedure.

SEE ALSO

For information about defining an action procedure, see "Defining Your Own Action Procedures" beginning on page 5-115.

GetControlVariant

To determine the variation code specified in the control definition function for a particular control, you can use the `GetControlVariant` function. The `GetControlVariant` function is also available as the `GetCVariant` function.

```
FUNCTION GetControlVariant (theControl: ControlHandle): Integer;
```

`theControl` A handle to the control whose variation code you wish to determine.

DESCRIPTION

The `GetControlVariant` function returns as its function result the variation code for the specified control.

SEE ALSO

Variation codes are described in "The Control Definition Function" on page 5-14.

GetAuxiliaryControlRecord

Use the `GetAuxiliaryControlRecord` function to get a handle to a control's auxiliary control record. The `GetAuxiliaryControlRecord` function is also available as the `GetAuxCtl` function.

```
FUNCTION GetAuxiliaryControlRecord (theControl: ControlHandle;
                                    VAR acHndl: AuxCtlHandle)
                                    : Boolean;
```

`theControl` A handle to a control.

`acHndl` A handle to the auxiliary control record for the control.

DESCRIPTION

In its `acHndl` parameter, the `GetAuxiliaryControlRecord` function returns a handle to the auxiliary control record for the specified control. Your application typically doesn't need to access an auxiliary control record unless you need its `acRefCon` field, which your application can use for any purpose.

The value that `GetAuxiliaryControlRecord` returns for a function result depends on the control's color control table, as described here:

- If your application has changed the default control color table for the given control (either by using the `SetControlColor` procedure or by creating its own control color table), the function returns `TRUE`.
- If your application has *not* changed the default control color table, the function returns `FALSE`.
- If you set the parameter `theControl` to `NIL`, the Dialog Manager ensures that the control uses the default color table, and `GetAuxiliaryControlRecord` returns `TRUE`.

Removing Controls

When you use the Window Manager procedures `DisposeWindow` and `CloseWindow` to remove a window, they automatically remove all controls associated with the window and release the memory the controls occupy.

When you no longer need a control in a window that you want to keep, you can use the `DisposeControl` procedure to remove the control from the window's control list and release the memory it occupies. You can use the `KillControls` procedure to dispose of all of a window's controls at once.

DisposeControl

To remove a particular control from a window that you want to keep, use the `DisposeControl` procedure.

```
PROCEDURE DisposeControl (theControl: ControlHandle);
```

`theControl` A handle to the control you wish to remove.

DESCRIPTION

The `DisposeControl` procedure removes the specified control from the screen, deletes it from its window's control list, and releases the memory occupied by the control record and any data structures associated with the control.

SPECIAL CONSIDERATIONS

The Window Manager procedures `CloseWindow` and `DisposeWindow` automatically dispose of all controls associated with the given window.

SEE ALSO

To remove all of the controls in a window, use the `KillControls` procedure, described next. The `CloseWindow` and `DisposeWindow` procedures are described in the chapter "Window Manager" in this book.

KillControls

To remove all of the controls in a particular window that you want to keep, use the `KillControls` procedure.

```
PROCEDURE KillControls (theWindow: WindowPtr);
```

`theWindow` A pointer to the window containing the controls to remove.

DESCRIPTION

The `KillControls` procedure disposes of all controls associated with the specified window by calling the `DisposeControl` procedure for each control.

SPECIAL CONSIDERATIONS

The Window Manager procedures `CloseWindow` and `DisposeWindow` automatically dispose of all controls associated with the given window.

SEE ALSO

The `CloseWindow` and `DisposeWindow` procedures are described in the chapter "Window Manager" in this book.

Application-Defined Routines

This section describes how to create your own control definition function—declared here as `MyControl`—which your application needs to provide when defining new, nonstandard controls. This section also describes action procedures—declared here as `MyAction` and `MyIndicatorAction`—which define additional actions to be performed repeatedly as long as the user holds down the mouse button while the cursor is in a control. For example, you need to define an action procedure for scrolling through a document while the user holds down the mouse button and the cursor is in a scroll arrow.

Defining Your Own Control Definition Function

In addition to the standard controls (buttons, checkboxes, radio buttons, pop-up menus, and scroll bars), the Control Manager allows you to define new, nonstandard controls as appropriate for your application. For example, you can define a three-way selector switch, a memory-space indicator that looks like a thermometer, or a thruster control for a spacecraft simulator. Controls and their indicators may occupy regions of any shape, as permitted by QuickDraw.

To define your own type of control, you write a control definition function, compile it as a resource of type `'CDEF'`, and store it in your resource file. (See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about creating resources.) Whenever you create a control, you specify a control definition ID, which the Control Manager uses to determine the control definition function. The control definition ID is an integer that contains the resource ID of the control definition function in its upper 12 bits and a variation code in its lower 4 bits. Thus, for a given resource ID and variation code

control definition ID = 16 x resource ID + variation code

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

You can define your own variation codes, which various Control Manager routines pass to your control definition function. This allows you to use one `'CDEF'` resource to handle several variations of the same general control.

The Control Manager calls the Resource Manager to access your control definition function with the given resource ID. The Resource Manager reads your control definition function into memory and returns a handle to it. The Control Manager stores this handle in the `contrlDefProc` field of the control record. In 24-bit addressing mode, the variation code is placed in the high-order byte of this field; in 32-bit mode, the variation code is placed in the most significant byte of the `acReserved` field in the control's `AuxCtlRec` record. Later, when various Control Manager routines need to perform a type-dependent action on the control, they call your control definition function and pass it the variation code as a parameter.

If you create a control definition function, you can use control color table records of any desired size and define their contents in any way you wish, except that part indices 1 through 127 are reserved for system definition. Note that in this case, you should allocate explicit auxiliary records for every control you create.

MyControl

If you wish to define new, nonstandard controls for your application, you must write a control definition function and store it in a resource file as a resource of type `'CDEF'`. Here's how you would declare a procedure named `MyControl`:

```
FUNCTION MyControl (varCode: Integer; theControl: ControlHandle;
                    message: Integer; param: LongInt): LongInt;
```

`varCode` The variation code for this control. To derive the control definition ID for the control, add this value to the result of 16 multiplied by the resource ID of the `'CDEF'` resource containing this function. The variation code allows you to specify several control definition IDs within one `'CDEF'` resource, thereby defining several variations of the same basic control.

`theControl` A handle to the control that the operation will affect.

`message` A value (from the following list) that specifies which operation your function must undertake.

```
CONST drawCntl       = 0;  {draw the control or its part}
      testCntl       = 1;  {test where mouse button }
                           { is pressed}
      calcCRgns      = 2;  {calculate region for }
                           { control or indicator in }
                           { 24-bit systems}
      initCntl       = 3;  {peform any additional }
                           { control initialization}
```

```
dispCntl       = 4;  {perform any additional }
                     { disposal actions}
posCntl        = 5;  {move indicator and }
                     { update its setting}
thumbCntl      = 6;  {calculate parameters for }
                     { dragging indicator}
dragCntl       = 7;  {perform any custom dragging }
                     { of control or its indicator}
autoTrack      = 8;  {execute action procedure }
                     { specified by your function}
calcCntlRgn    = 10; {calculate region for control}
calcThumbRgn   = 11; {calculate region for }
                     { indicator}
```

`param` A value whose meaning depends on the operation specified in the `message` parameter.

DESCRIPTION

The Control Manager calls your control definition function under various circumstances; the Control Manager uses the `message` parameter to inform your control definition function what action it must perform. The data that the Control Manager passes in the `param` parameter, the action that your control definition function must undertake, and the function result that your control definition function returns all depend on the value that the Control Manager passes in the `message` parameter. The rest of this section describes how to respond to the various values that the Control Manager passes in the `message` parameter.

Drawing the Control or Its Part

When the Control Manager passes the value for the `drawCntl` constant in the `message` parameter, the low word in the `param` parameter has one of the following values:

- the value 0, indicating the entire control
- the value 129, signifying an indicator that must be moved
- any other value, indicating a part code for the control (Don't use part code 128, which is reserved for future use, or part code 129, which the Control Manager uses to signify an indicator that must be moved.)

Note
For the `drawCntl` message, the high-order word of the `param` parameter may contain undefined data; therefore, evaluate only the low-order word of this parameter. ◆

If the specified control is visible, your control definition function should draw the control (or the part specified in the `param` parameter) within the control's rectangle. If the control is invisible (that is, if its `contrlVis` field is set to 0), your control definition function does nothing.

When drawing the control or its part, take into account the current values of its `contrlHilite` and `contrlValue` fields of the control's control record.

If the part code for your control's indicator is passed in `param`, assume that the indicator hasn't moved; the Control Manager, for example, may be calling your control definition function so that you may simply highlight the indicator. However, when your application calls the `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum` procedures, they in turn may call your control definition function to redraw the indicator. Since these routines have no way of determining what part code you chose for your indicator, they all pass 129 in `param`, meaning that you should move your indicator. Your control definition function must detect this part code as a special case and remove the indicator from its former location before drawing it. If your control has more than one indicator, you should interpret 129 to mean all indicators.

When passed the value for the `drawCntl` constant in the `message` parameter, your control definition function should always return 0 as its function result.

Testing Where the Mouse-Down Event Occurs

To request your control definition function to determine whether a specified point is in a visible control, the `FindControl` function sends the value for the `testCntl` constant in the `message` parameter. In this case, the `param` parameter specifies a point (in coordinates local to the control's window) as follows:

- The point's vertical coordinate is contained in the high-order word of the long integer.
- The point's horizontal coordinate is contained in the low-order word.

When passed the value for the `testCntl` constant in the `message` parameter, your control definition function should return the part code of the part that contains the specified point; it should return 0 if the point is outside the control or if the control is inactive.

Calculating the Control and Indicator Regions

When the Control Manager passes the value for the `calcCRgns` constant in the `message` parameter, your control definition function should calculate the region occupied by either the control or its indicator. The Control Manager passes a QuickDraw region handle in the `param` parameter; it is this region that you calculate. If the high-order bit of `param` is set, the region requested is that of the control's indicator; otherwise, the region requested is that of the entire control. Your control definition function should clear the high bit of the region handle before calculating the region.

When the Control Manager passes the value for the `calcCntlRgn` constant in the `message` parameter, your control definition function should calculate the region passed in the `param` parameter for the specified control. When the Control Manager passes the value for the `calcThumbRgn` constant, calculate the region occupied by the indicator.

When passed the values for the `calcCRgns`, `calcCntlRgn`, and `calcThumbRgn` constants, your control definition function should always return 0, and it should express the region in the local coordinate system of the control's window.

IMPORTANT
The Control Manager passes the `calcCRgns` constant when the 24-bit Memory Manager is in operation. When the 32-bit Memory Manager is in operation, the Control Manager instead passes the `calcCntlRgn` constant or the `calcThumbRgn` constant. Your control definition function should respond to all three constants. ▲

Performing Any Additional Initialization

After initializing fields of a control record as appropriate when creating a new control, the Control Manager passes `initCntl` in the `message` parameter to give your control definition function the opportunity to perform any type-specific initialization you may require. For example, if you implement the control's action procedure in its control definition function, you'll need to store `Pointer(-1)` in the `contrlAction` field of the control's control record. Then, in a call to `TrackControl` for this control, you would pass `Pointer(-1)` in the `actionProc` parameter of `TrackControl`.

The standard control definition function for scroll bars allocates space for a region to hold the scroll box and stores the region handle in the `contrlData` field of the new control record.

When passed the value for the `initCntl` constant in the `message` parameter, your control definition function should ignore the `param` parameter and return 0 as a function result.

Performing Any Additional Disposal Actions

The `DisposeControl` procedure passes `dispCntl` in the `message` parameter to give your control definition function the opportunity to carry out any additional actions when disposing of a control. For example, the standard definition function for scroll bars releases the memory occupied by the scroll box region, whose handle is kept in the `contrlData` field of the control's control record.

When passed the value for the `dispCntl` constant in the `message` parameter, your control definition function should ignore the `param` parameter and return 0 as a function result.

Moving the Indicator

When a mouse-up event occurs in the indicator of a control, the `TrackControl` function calls your control definition function and passes `posCntl` in the `message` parameter. In this case, the `param` parameter contains a point (in coordinates local to the control's window) that specifies the vertical and horizontal offset, in pixels, by which your control definition function should move the indicator from its current position. Typically, this is the offset between the points where the cursor was when the user pressed and released the mouse button while dragging the indicator. The offset point is specified as follows:

- The point's vertical offset is contained in the high-order word of the `param` parameter.
- The point's horizontal offset is contained in the low-order word.

Your definition function should calculate the control's new setting based on the given offset and then, to reflect the new setting, redraw the control and update the `contrlValue` field in the control's control record. Your control definition function should ignore the `param` parameter and return 0 as a function result.

Note that the `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum` procedures do not call your control definition function with the `posCntl` message; instead, they pass the `drawCntl` message.

Calculating Parameters for Dragging the Indicator

When the Control Manager passes the value for `thumbCntl` in the `message` parameter, your control definition function should respond by calculating values (analogous to the `limitRect`, `slopRect`, and `axis` parameters of `DragControl`) that constrain how the indicator is dragged. The `param` parameter contains a pointer to the following data structure:

```
RECORD
   limitRect,slopRect:  Rect;
   axis:                Integer;
END;
```

On entry, the field `param^.limitRect.topLeft` contains the point where the mouse-down event first occurred. Your definition function should store the appropriate values into the fields of the record pointed to by `param`; they're analogous to the similarly named parameters to the Window Manager function `DragGrayRgn`.

Performing Custom Dragging

The Control Manager passes `dragCntl` in the `message` parameter to give your control definition function the opportunity to specify its own method for dragging a control (or its indicator).

The `param` parameter specifies whether the user is dragging an indicator or the whole control:

- A value of 0 means the user is dragging the entire control.
- Any nonzero value means the user is dragging only the indicator.

If you want to use the Control Manager's default method of dragging (which is to call `DragControl` to drag the control or the Window Manager function `DragGrayRgn` to drag its indicator), return 0 as the function result for your control definition function.

If your control definition function returns any nonzero result, the Control Manager does not drag your control, and instead your control definition function must drag the specified control (or its indicator) to follow the cursor until the user releases the mouse button, as follows:

- If the user drags the entire control, your definition function should use the `MoveControl` procedure to reposition the control to its new location after the user releases the mouse button.

- If the user drags the indicator, your definition function must calculate the control's new setting (based on the pixel offset between the points where the cursor was when the user pressed and released the mouse button while dragging the indicator) and then, to reflect the new setting, redraw the control and update the `contrlValue` field in the control's control record. Note that, in this case, the `TrackControl` function returns 0 whether or not the user changes the indicator's position. Thus, you must determine whether the user has changed the control's setting, for instance, by comparing the control's value before and after the call to `TrackControl`.

Executing an Action Procedure

You can design a control whose action procedure is specified by your control definition function. When you create the control, your control definition function must first respond to the `initCntl` message by storing `Pointer(-1)` in the `contrlAction` field of the control's control record. (As previously explained, the Control Manager sends the `initCntl` message to your control definition function after initializing the fields of a new control record.) Then, when your application passes `Pointer(-1)` in the `actionProc` parameter to the `TrackControl` function, `TrackControl` calls your control definition function with the `autoTrack` message. The `param` parameter specifies the part code of the part where the mouse-down event occurs. Your control definition function should then use this information to respond as an action procedure would.

Note
For the `autoTrack` message, the high-order word of the `param` parameter may contain undefined data; therefore, evaluate only the low-order word of this parameter. ◆

ASSEMBLY-LANGUAGE INFORMATION

The function's entry point must be at the beginning.

SEE ALSO

The `TrackControl` function is described on page 5-90; creating an action procedure is described in the next section.

Defining Your Own Action Procedures

When a mouse-down event occurs in a control, the `TrackControl` function responds as appropriate by highlighting the control or dragging the indicator as long as the user holds down the mouse button. You can define other actions to be performed repeatedly during this interval. To do so, define your own action procedure and point to it in the `actionProc` parameter of the `TrackControl` function.

When calling your action procedure for a control part other than an indicator, `TrackControl` passes your action procedure (1) a handle to the control and (2) the control's part code. Your action procedure should then respond as appropriate. For

example, if the user is working in a text document and holds down the mouse button while the cursor is in the lower scroll arrow, your application should scroll continuously one line at a time until the user releases the mouse button or reaches the end of the document.

For a control part other than an indicator, you declare an action procedure that takes two parameters: a handle to the control in which the mouse-down event occurred and an integer that represents the part of the control in which the mouse-down event occurred. Such an action procedure is declared as `MyAction` in the following section.

If the mouse-down event occurs in an indicator, your action procedure should take no parameters, because the user may move the cursor outside the indicator while dragging it. Such an action procedure, declared here as `MyIndicatorAction`, is described on page 5-117.

Because it will be called with either zero or two parameters, according to whether the mouse-down event occurred in an indicator or elsewhere, your action procedure can be defined for only one case or the other. The only way to specify actions in response to all mouse-down events in a control, regardless of whether they're in an indicator, is to define your own control definition function, as described in "Defining Your Own Control Definition Function" beginning on page 5-109.

MyAction

Here's how to declare an action procedure for a control part other than an indicator if you were to name the procedure `MyAction`:

```
PROCEDURE MyAction (theControl: ControlHandle; partCode: Integer);
```

`theControl` A handle to the control in which the mouse-down event occurred.

`partCode` When the cursor is still in the control part where mouse-down event first occurred, this parameter contains that control's part code. When the user drags the cursor outside the original control part, this parameter contains 0.

DESCRIPTION

Your procedure can perform any action appropriate for the control part. For example, when a mouse-down event occurs in a scroll arrow or gray area of a scroll bar, `TrackControl` calls your action procedure and passes it the part code and a handle to the scroll bar. Your action procedure should examine the part code to determine the part of the control in which the mouse-down event occurred. Your action procedure should then scroll up or down a line or page as appropriate and then call the `SetControlValue` procedure to change the control's setting and redraw the scroll box.

ASSEMBLY-LANGUAGE INFORMATION

If you store a pointer to a procedure in the global variable `DragHook`, your procedure is called repeatedly (with no parameters) as long as the user holds down the mouse button. The `TrackControl` function invokes the Window Manager function `DragGrayRgn`, which calls the `DragHook` procedure. The `DragGrayRgn` function uses the pattern stored in the global variable `DragPattern` for the dragged outline of the indicator.

SEE ALSO

Listing 5-19 on page 5-59 illustrates a pair of action procedures for scrolling through a text document. As an alternative to passing a pointer to your action procedure in a parameter to `TrackControl`, you can use the `SetControlAction` procedure to store a pointer to the action procedure in the `contrlAction` field in the control record. When you pass `Pointer(–1)` instead of a procedure pointer to `TrackControl`, `TrackControl` uses the action procedure pointed to in the control record.

MyIndicatorAction

Here's how to declare an action procedure for an indicator if you were to name the procedure `MyIndicatorAction`:

```
PROCEDURE MyIndicatorAction;
```

DESCRIPTION

Your procedure can perform any action appropriate for the control part. For example, if your application plays music while displaying a volume control slider, your application should change the volume in response to the user's action in the slider switch.

SEE ALSO

See the `MyAction` procedure described on page 5-116 for other considerations.

Resources

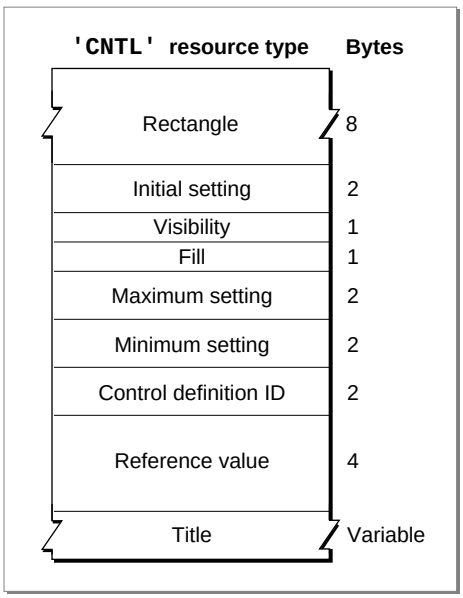
This section describes the control (`'CNTL'`) resource and the control color table (`'cctb'`) resource. You can use the control resource to define a control and use the control color table resource to change the default colors of a control's parts.

The Control Resource

You can use a control resource to define a control. A control resource is a resource of type `'CNTL'`. All control resources must have resource ID numbers greater than 128. Use the `GetNewControl` function (described on page 5-81) to create a control defined in a control resource. The Control Manager uses the information you specify to create a control record in memory. (The control record is described on page 5-73.)

This section describes the structure of this resource after it is compiled by the Rez resource compiler, available from APDA. The format of a Rez input file for a control resource differs from its compiled output form, which is illustrated in Figure 5-25. If you are concerned only with creating a control resource, see "Creating and Displaying a Control" beginning on page 5-15.

Figure 5-25 Structure of a compiled control (`'CNTL'`) resource



The compiled version of a control resource contains the following elements:

- The rectangle, specified in coordinates local to the window, that encloses the control; this rectangle encloses the control and thus determines its size and location.
- The initial setting for the control.
  - For controls—such as buttons—that don't retain a setting, this value should be 0.
  - For controls—such as checkboxes or radio buttons—that retain an on-or-off setting, a value of 0 in this element indicates that the control is initially off; a value of 1 indicates that the control is initially on.
  - For controls—such as scroll bars and dials—that can take a range of settings, whatever initial value is appropriate within that range is specified in this element.

- For pop-up menus, a combination of values instructs the Control Manager where and how to draw the control title. Appropriate values, along with the constants used to specify them in a Rez input file, are listed here:

```
CONST popupTitleBold       = $00000100;   {boldface font style}
      popupTitleItalic     = $00000200;   {italic font style}
      popupTitleUnderline  = $00000400;   {underline font }
                                          { style}
      popupTitleOutline    = $00000800;   {outline font style}
      popupTitleShadow     = $00001000;   {shadow font style}
      popupTitleCondense   = $00002000;   {condensed text}
      popupTitleExtend     = $00004000;   {extended text}
      popupTitleNoStyle    = $00008000;   {monostyle text}
      popupTitleLeftJust   = $00000000;   {place title left }
                                          { of pop-up box}
      popupTitleCenterJust = $00000001;   {center title over }
                                          { pop-up box}
      popupTitleRightJust  = $000000FF;   {place title right }
                                          { of pop-up box}
```

- The visibility of the control. If this element contains the value `TRUE`, `GetNewControl` draws the control immediately, without using the application's standard updating mechanism for windows. If this element contains the value `FALSE`, the application must use the `ShowControl` procedure (described on page 5-86) when it's prepared to display the control.
- Fill. This should be set to 0.
- The maximum setting for the control.
  - For controls—such as buttons—that don't retain a setting, this value should be 1.
  - For controls—such as checkboxes or radio buttons—that retain an on-or-off setting, this element should contain the value 1 (meaning "on").
  - For controls—such as scroll bars and dials—that can take a range of settings, this element can contain whatever maximum value is appropriate; when the application makes the maximum setting of a scroll bar equal to its minimum setting, the control definition function automatically makes the scroll bar inactive, and when the application makes the maximum setting exceed the minimum, the control definition function makes the scroll bar active again.
  - For pop-up menus, this element contains the width, in pixels, of the control title.
- The minimum setting for the control.
  - For controls—such as buttons—that don't retain a setting, this value should be 0.
  - For controls—such as checkboxes or radio buttons—that retain an on-or-off setting, the value 0 (meaning "off") should be set in this element.
  - For controls—such as scroll bars and dials—that can take a range of settings, this element contains whatever minimum value is appropriate.
  - For pop-up menus, this element contains the resource ID of the `'MENU'` resource that describes the menu items.

- The control definition ID, which the Control Manager uses to determine the control definition function for this control. "Defining Your Own Control Definition Function" beginning on page 5-109 describes how to create control definition functions and their corresponding control definition IDs. The following list shows the control definition ID numbers—and the constants that represent them in Rez input files—for the standard controls.

```
CONST
   pushButProc          = 0;     {button}
   checkBoxProc         = 1;     {checkbox}
   radioButProc         = 2;     {radio button}
   useWFont             = 8;     {when added to above, shows }
                                 { title in the window font}
   scrollBarProc        = 16;    {scroll bar}
   popupMenuProc        = 1008;  {pop-up menu}
   popupFixedWidth      = $0001; {add to popupMenuProc to }
                                 { use fixed-width control}
   popupUseAddResMenu   = $0004; {add to popupMenuProc to }
                                 { specify a value of type }
                                 { ResType in the contrlRfCon }
                                 { field of the control }
                                 { record; Menu Manager }
                                 { adds resources of this }
                                 { type to the menu}
   popupUseWFont        = $0008; {if added to popupMenuProc, }
                                 { shows title in window font}
```

Note

The title of a button, checkbox, radio button, or pop-up menu normally appears in the system font, which in Roman script systems is 12-point Chicago. Do not use a smaller font; some script systems, such as KanjiTalk, require 12-point fonts. You should generally use the system font in your controls; doing so will simplify localization effort. However, if you absolutely need to display a control title in the font currently associated with the window's graphics port, you can add the `popupUseWFont` constant to the pop-up menu control definition ID or add the `useWFont` constant to the other standard control definition IDs. ◆

- The control's reference value, which is set and used only by the application (except when the application adds the `popupUseAddResMenu` variation code to the `popupMenuProc` control definition ID, as described in "Creating a Pop-Up Menu" beginning on page 5-25).

- For controls—such as buttons, checkboxes, radio buttons, and pop-up menus—that need a title, the string for that title; for controls that don't use titles, an empty string.

After you use `GetNewControl` to create the control, you can change the current setting, the maximum setting, the minimum setting, the reference value, and the title by using, respectively, the `SetControlValue`, `SetControlMaximum`, `SetControlMinimum`, `SetControlReference`, and `SetControlTitle` routines. You can use the `MoveControl` and `SizeControl` procedures to change the control's rectangle. You can use the `GetControlValue`, `GetControlMaximum`, `GetControlMinimum`, `GetControlReference`, and `GetControlTitle` routines to determine the control values.

The Control Color Table Resource

On color monitors, the Control Manager automatically draws control parts so that they match the colors of the controls used by system software.

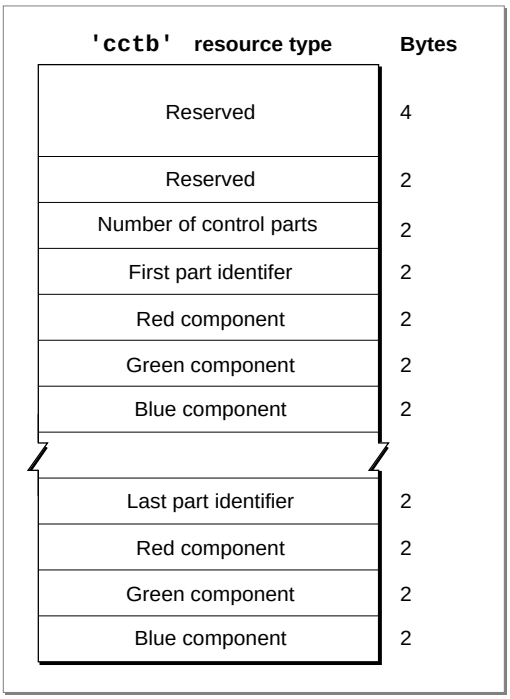
If you feel absolutely compelled to use nonstandard colors, the Control Manager allows you to do so. Your application can specify these by creating a control color table (`'cctb'`) resource; you must give the control color table resource for a control the same resource ID as its control (`'CNTL'`) resource, which is described on page 5-118. When you call the `GetNewControl` function to create the control, the Control Manager automatically attempts to load a control color table resource with the same resource ID as the control resource specified to `GetNewControl`. The Control Manager also creates an auxiliary control record for the control; the auxiliary control record is described on page 5-76.

Note
Using nonstandard colors in your controls may initially confuse your users. ◆

Generally, you use a control color table resource for a control that you define in a control resource. To change a control's colors, or to use nonstandard colors in a control you create using `NewControl`, create a control color table record and use the `SetControlColor` procedure. The control color table record is described on page 5-77; the `SetControlColor` procedure is described on page 5-101.

A control color table resource is of type `'cctb'`. All control color table resources must have resource ID numbers greater than 128. Figure 5-26 on the next page shows the format of a control color table resource. Note that `DisposeControl` does not delete a control color table resource; therefore, you should make each control color table resource purgeable.

Figure 5-26 Structure of a compiled control color table (`'cctb'`) resource



You define a control color table resource by specifying these elements in a resource with the `'cctb'` resource type:

- Reserved. Should always be set to 0.
- Reserved. Should always be set to 0.
- Number of control parts. For standard controls other than scroll bars, this should be set to 3, because these controls consist of a frame, a control body, and text. For scroll bars, this should be set to 12. A scroll bar consists of a frame, a body, and scroll box; each part of a scroll bar has various highlight and tinge colors associated with it. To create a control with more parts, you must create your own control definition function (as described in "Defining Your Own Control Definition Function" beginning on page 5-109) that recognizes additional parts.
- First part identifier. A value or constant that identifies the control's part to color. The part identifiers can be listed in any order. The scroll bar control definition function may use more than one part identifier to produce the actual colors used for each part of the scroll bar.

```
CONST
   cFrameColor       = 0;  {frame color; for scroll bars, used to produce }
                           { foreground color for scroll arrows & gray area}
   cBodyColor        = 1;  {body color; for scroll bars, used to produce }
                           { colors in the scroll box}
   cTextColor        = 2;  {text color; unused for scroll bars}
```

```
cArrowsColorLight = 5;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cArrowsColorDark  = 6;  {Used to produce colors in arrows & scroll bar }
                        { background color}
cThumbLight       = 7;  {Used to produce colors in scroll box}
cThumbDark        = 8;  {Used to produce colors in scroll box}
cHiliteLight      = 9;  {Use same value as wHiliteColorLight in 'wctb'}
cHiliteDark       = 10; {Use same value as wHiliteColorDark in 'wctb'}
cTitleBarLight    = 11; {Use same value as wTitleBarLight in 'wctb'}
cTitleBarDark     = 12; {Use same value as wTitleBarDark in 'wctb'}
cTingeLight       = 13; {Use same value as wTingeLight in 'wctb'}
cTingeDark        = 14; {Use same value as wTingeDark in 'wctb'}
```

- Red component. An integer that represents the intensity of the red component of the color to use when drawing this part of the control. In this and the next two elements, use 16-bit unsigned integers to give the intensity values of three additive primary colors.
- Green component. An integer that represents the intensity of the green component of the color to use when drawing this part of the control.
- Blue component. An integer that represents the intensity of the blue component of the color to use when drawing this part of the control.
- Part identifier and red, green, and blue components for the next control part. You can list parts in any order in this resource. If the application specifies a part identifier that cannot be found, the Control Manager uses the colors for the control's first identifiable part. If a part is not listed in the control color table, the Dialog Manager draws it in its default color.

The Control Definition Function

The resource type for a control definition function is `'CDEF'`. The resource data is the compiled or assembled code of the function. See "Defining Your Own Control Definition Function" beginning on page 5-109 for information about creating a control definition function.

Summary of the Control Manager

Pascal Summary

Constants

```
CONST
   {control definition IDs}
   pushButProc          = 0;     {button}
   checkBoxProc         = 1;     {checkbox}
   radioButProc         = 2;     {radio button}
   useWFont             = 8;     {add to above to display control title in }
                                 { the window font}
   scrollBarProc        = 16;    {scroll bar}
   popupMenuProc        = 1008;  {pop-up menu}
   popupMenuCDEFproc    = popupMenuProc;  {synonym for compatibility}

   {pop-up menu CDEF variation codes}
   popupFixedWidth      = $0001;    {add to popupMenuProc to use }
                                    { fixed-width control}
   popupUseAddResMenu   = $0004;    {add to popupMenuProc to specify a }
                                    { value of type ResType in the }
                                    { contrlRfCon field of the control }
                                    { record; Menu Manager adds }
                                    { resources of this type to the menu}
   popupUseWFont        = $0008;    {add to popupMenuProc to show control }
                                    { title in the window font}

   {part codes}
   inButton             = 10;    {button}
   inCheckBox           = 11;    {checkbox or radio button}
   inUpButton           = 20;    {up arrow for a vertical scroll bar, }
                                 { left arrow for a horizontal scroll bar}
   inDownButton         = 21;    {down arrow for a vertical scroll bar, }
                                 { right arrow for a horizontal scroll bar}
   inPageUp             = 22;    {gray area above scroll box for a }
                                 { vertical scroll bar, gray area to }
                                 { left of scroll box for a horizontal }
                                 { scroll bar}
```

```
inPageDown           = 23;    {gray area below scroll box for a }
                              { vertical scroll bar, gray area to }
                              { right of scroll box for a horizontal }
                              { scroll bar}
inThumb              = 129;   {scroll box (or other indicator)}

{pop-up title characteristics}
popupTitleBold       = $00000100;   {boldface font style}
popupTitleItalic     = $00000200;   {italic font style}
popupTitleUnderline  = $00000400;   {underline font style}
popupTitleOutline    = $00000800;   {outline font style}
popupTitleShadow     = $00001000;   {shadow font style}
popupTitleCondense   = $00002000;   {condensed characters}
popupTitleExtend     = $00004000;   {extended characters}
popupTitleNoStyle    = $00008000;   {monostyled text}
popupTitleLeftJust   = $00000000;   {place title left of pop-up box}
popupTitleCenterJust = $00000001;   {center title over pop-up box}
popupTitleRightJust  = $000000FF;   {place title right of pop-up box}

{axis constraints for DragControl procedure}
noConstraint         = 0;  {no constraint}
hAxisOnly            = 1;  {drag along horizontal axis only}
vAxisOnly            = 2;  {drag along vertical axis only}

{constants for the message parameter in a control definition function}
drawCntl             = 0;  {draw the control or its part}
testCntl             = 1;  {test where mouse button is pressed}
calcCRgns            = 2;  {calculate region for control or indicator in }
                           { 24-bit systems}
initCntl             = 3;  {peform any additional control initialization}
dispCntl             = 4;  {take any additional disposal actions}
posCntl              = 5;  {move indicator and update its setting}
thumbCntl            = 6;  {calculate parameters for dragging indicator}
dragCntl             = 7;  {perform any custom dragging of control or }
                           { its indicator}
autoTrack            = 8;  {execute action procedure specified by your }
                           { function}
calcCntlRgn          = 10; {calculate region for control}
calcThumbRgn         = 11; {calculate region for indicator}

{part identifiers for ColorSpec records in a control color table resource}
cFrameColor          = 0;  {frame color; for scroll bars, also fore- }
                           { ground color for scroll arrows and gray area}
```

```
   cBodyColor           = 1;  {for scroll bars, background color for }
                              { scroll arrows and gray area; for other }
                              { controls, the fill color for body of control}
   cTextColor           = 2;  {text color; unused for scroll bars}
   cThumbColor          = 3;  {Reserved}
```

Data Types

```
TYPE  ControlPtr        = ^ControlRecord;
      ControlHandle     = ^ControlPtr;

      ControlRecord =
      PACKED RECORD
         nextControl:   ControlHandle; {next control}
         contrlOwner:   WindowPtr;     {control's window}
         contrlRect:    Rect;          {rectangle}
         contrlVis:     Byte;          {255 if visible}
         contrlHilite:  Byte;          {highlight state}
         contrlValue:   Integer;       {control's current setting}
         contrlMin:     Integer;       {control's minimum setting}
         contrlMax:     Integer;       {control's maximum setting}
         contrlDefProc: Handle;        {control definition function}
         contrlData:    Handle;        {data used by contrlDefProc}
         contrlAction:  ProcPtr;       {action procedure}
         contrlRfCon:   LongInt;       {control's reference value}
         contrlTitle:   Str255;        {control's title}
      END;

      AuxCtlPtr      = ^AuxCtlRec;
      AuxCtlHandle   = ^AuxCtlPtr;

      AuxCtlRec =
      RECORD
         acNext:        AuxCtlHandle;  {handle to next AuxCtlRec}
         acOwner:       ControlHandle; {handle to this record's control}
         acCTable:      CCTabHandle;   {handle to color table record}
         acFlags:       Integer;       {reserved}
         acReserved:    LongInt;       {reserved for future use}
         acRefCon:      LongInt;       {for use by application}
      END;
```

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

CtlCTab =
RECORD
   ccSeed:        LongInt;       {reserved; set to 0}
   ccRider:       Integer;       {reserved; set to 0}
   ctSize:        Integer;       {number of ColorSpec records in next }
                                 { field; 3 for standard controls}
   ctTable:       ARRAY[0..3] OF ColorSpec;
END;
```

Control Manager Routines

Creating Controls

```
FUNCTION GetNewControl      (controlID: Integer; owner: WindowPtr)
                             : ControlHandle;
FUNCTION NewControl         (theWindow: WindowPtr; boundsRect: Rect;
                             title: Str255; visible: Boolean;
                             value: Integer; min: Integer; max: Integer;
                             procID: Integer; refCon: LongInt)
                             : ControlHandle;
```

Drawing Controls

```
{UpdateControls is also spelled as UpdtControl}
PROCEDURE ShowControl       (theControl: ControlHandle);
PROCEDURE UpdateControls    (theWindow: WindowPtr; updateRgn: RgnHandle);
PROCEDURE DrawControls      (theWindow: WindowPtr);
PROCEDURE Draw1Control      (theControl: ControlHandle);
```

Handling Mouse Events in Controls

```
FUNCTION FindControl        (thePoint: Point; theWindow: WindowPtr;
                             VAR theControl: ControlHandle): Integer;
FUNCTION TrackControl       (theControl: ControlHandle; thePoint: Point;
                             actionProc: ProcPtr): Integer;
FUNCTION TestControl        (theControl: ControlHandle; thePt: Point)
                             : Integer;
```

Changing Control Settings and Display

```
{some routines have 2 spellings—see Table 5-1 for the alternate spellings}
PROCEDURE SetControlValue   (theControl: ControlHandle; theValue: Integer);
PROCEDURE SetControlMinimum (theControl: ControlHandle; minValue: Integer);
PROCEDURE SetControlMaximum (theControl: ControlHandle; maxValue: Integer);
PROCEDURE SetControlTitle   (theControl: ControlHandle; title: Str255);
PROCEDURE HideControl       (theControl: ControlHandle);
PROCEDURE MoveControl       (theControl: ControlHandle; h: Integer;
                             v: Integer);
PROCEDURE SizeControl       (theControl: ControlHandle; w: Integer; h:
                             Integer);
PROCEDURE HiliteControl     (theControl: ControlHandle;
                             hiliteState: Integer);
PROCEDURE DragControl       (theControl: ControlHandle; startPt: Point;
                             limitRect: Rect; slopRect: Rect;
                             axis: Integer);
PROCEDURE SetControlColor   (theControl: ControlHandle; newColorTable:
                             CCTabHandle);
PROCEDURE SetControlAction  (theControl: ControlHandle;
                             actionProc: ProcPtr);
```

Determining Control Values

```
{some routines have 2 spellings—see Table 5-1 for the alternate spellings}
FUNCTION GetControlValue    (theControl: ControlHandle): Integer;
FUNCTION GetControlMinimum  (theControl: ControlHandle): Integer;
FUNCTION GetControlMaximum  (theControl: ControlHandle): Integer;
PROCEDURE GetControlTitle   (theControl: ControlHandle; VAR title: Str255);
FUNCTION GetControlReference
                            (theControl: ControlHandle): LongInt;
PROCEDURE SetControlReference
                            (theControl: ControlHandle; data: LongInt);
FUNCTION GetControlAction   (theControl: ControlHandle): ProcPtr;
FUNCTION GetControlVariant  (theControl: ControlHandle): Integer;
FUNCTION GetAuxiliaryControlRecord
                            (theControl: ControlHandle;
                             VAR acHndl: AuxCtlHandle): Boolean;
```

Removing Controls

```
PROCEDURE DisposeControl    (theControl: ControlHandle);
PROCEDURE KillControls      (theWindow: WindowPtr);
```

Application-Defined Routines

Defining Your Own Control Definition Function

```
FUNCTION MyControl          (varCode: Integer; theControl: ControlHandle;
                             message: Integer; param: LongInt) : LongInt;
```

Defining Your Own Action Procedures

```
PROCEDURE MyAction          (theControl: ControlHandle; partCode: Integer);
PROCEDURE MyIndicatorAction;
```

C Summary

Constants

```
enum {
      /*control definition IDs*/
      pushButProc          = 0,     /*button*/
      checkBoxProc         = 1,     /*checkbox*/
      radioButProc         = 2,     /*radio button*/
      useWFont             = 8,     /*add to above to display control */
                                    /* title in the window font*/
      scrollBarProc        = 16,    /*scroll bar*/
      popupMenuProc        = 1008,  /*pop-up menu*/

      /*pop-up menu CDEF variation codes*/
      popupFixedWidth  = 1 << 0,    /*add to popupMenuProc to use */
                                    /* use fixed-width control*/
      popupUseAddResMenu = 1 << 2,  /*add to popupMenuProc to specify a */
                                    /* value of type ResType in the */
                                    /* contrlRfCon field of the control */
                                    /* record; Menu Manager adds */
                                    /* resources of this type to the menu*/
      popupUseWFont  = 1 << 3       /*add to popupMenuProc to display */
                                    /* control title in the window font*/
};
```

```
enum {
      /*part codes*/
      inButton             = 10, /*button*/
      inCheckBox           = 11, /*checkbox or radio button*/
      inUpButton           = 20, /*up arrow for a vertical scroll bar, */
                                 /* left arrow for a horizontal scroll bar*/
      inDownButton         = 21, /*down arrow for a vertical scroll bar, */
                                 /* right arrow for a horizontal scroll bar*/
      inPageUp             = 22, /*gray area above scroll box for a */
                                 /* vertical scroll bar, gray area to */
                                 /* left of scroll box for a horizontal */
                                 /* scroll bar*/
      inPageDown           = 23, /*gray area below scroll box for a */
                                 /* vertical scroll bar, gray area to */
                                 /* right of scroll box for a horizontal */
                                 /* scroll bar*/
      inThumb              = 129 /*scroll box (or other indicator)*/
};

enum {
      /*pop-up title characteristics*/
      popupTitleBold  = 1 << 8,        /*boldface font style*/
      popupTitleItalic = 1 << 9,       /*italic font style*/
      popupTitleUnderline = 1 << 10,   /*underline font style*/
      popupTitleOutline = 1 << 11,     /*outline font style*/
      popupTitleShadow = 1 << 12,      /*shadow font style*/
      popupTitleCondense = 1 << 13,    /*condensed text*/
      popupTitleExtend = 1 << 14,      /*extended text*/
      popupTitleNoStyle = 1 << 15      /*monostyled text*/
};

enum {
      /*pop-up title characteristics*/
      popupTitleLeftJust = 0x00000000,    /*place title left of pop-up box*/
      popupTitleCenterJust = 0x00000001,  /*center title over pop-up box*/
      popupTitleRightJust = 0x000000FF,   /*place title right of pop-up box*/

   /*axis constraints for DragControl procedure*/
      noConstraint   = 0,  /*no constraint*/
      hAxisOnly      = 1,  /*constrain movement to horizontal axis only*/
      vAxisOnly      = 2,  /*constrain movement to vertical axis only*/
```

```
/*constants for the message parameter in a control definition function*/
      drawCntl       = 0,  /*draw the control or control part*/
      testCntl       = 1,  /*test where mouse button was pressed*/
      calcCRgns      = 2,  /*calculate region for control or indicator in */
                           /* 24-bit systems*/
      initCntl       = 3,  /*do any additional control initialization*/
      dispCntl       = 4,  /*take any additional disposal actions*/
      posCntl        = 5,  /*move indicator and update its setting*/
      thumbCntl      = 6,  /*calculate parameters for dragging indicator*/
      dragCntl       = 7,  /*peform any custom dragging of control or */
                           /* its indicator*/
      autoTrack      = 8,  /*execute action procedure specified by your */
                           /* function*/
      calcCntlRgn    = 10, /*calculate region for control*/
      calcThumbRgn   = 11, /*calculate region for indicator*/

/*part identifiers for ColorSpec records in a control color table resource*/
      cFrameColor    = 0,  /*frame color; for scroll bars, also foreground */
                           /* color for scroll arrows and gray area*/
      cBodyColor     = 1,  /*for scroll bars, background color for scroll */
                           /* arrows and gray area; for other controls, */
                           /* the fill color for body of control*/
      cTextColor     = 2,  /*text color; for scroll bars, unused*/
      cThumbColor    = 3   /*Reserved*/
};
```

Data Types

```
struct ControlRecord {
      struct ControlRecord **nextControl;    /*next control*/
      WindowPtr      contrlOwner;   /*control's window*/
      Rect           contrlRect;    /*rectangle*/
      unsigned char  contrlVis;     /*255 if visible*/
      unsigned char  contrlHilite;  /*highlight state*/
      short          contrlValue;   /*control's current setting*/
      short          contrlMin;     /*control's minimum setting*/
      short          contrlMax;     /*control's maximum setting*/
      Handle         contrlDefProc; /*control definition function*/
      Handle         contrlData;    /*data used by contrlDefProc*/
      ProcPtr        contrlAction;  /*action procedure*/
      long           contrlRfCon;   /*control's reference value*/
      Str255         contrlTitle;   /*control's title*/
   };
```

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

struct AuxCtlRec {
      Handle         acNext;        /*handle to next AuxCtlRec*/
      ControlHandle  acOwner;       /*handle to this record's control*/
      CCTabHandle    acCTable;      /*handle to color table record*/
      short          acFlags;       /*reserved*/
      long           acReserved;    /*reserved for future use*/
      long           acRefCon;      /*for use by application*/
   };

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

struct CtlCTab {
      long           ccSeed;        /*reserved; set to 0*/
      short          ccRider;       /*reserved; set to 0*/
      short          ctSize;        /*number of ColorSpec records in next */
                                    /* field; 3 for standard controls*/
      ColorSpec      ctTable[4];
   };
typedef struct CtlCTab CtlCTab;
typedef CtlCTab *CCTabPtr, **CCTabHandle;
```

Control Manager Routines

Creating Controls

```
pascal ControlHandle GetNewControl
                            (short controlID, WindowPtr owner);
pascal ControlHandle NewControl
                            (WindowPtr theWindow, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short value, short min, short max,
                             short procID, long refCon);
```

Drawing Controls

```
/*UpdateControls is also spelled as UpdtControl*/
pascal void ShowControl     (ControlHandle theControl);
pascal void UpdateControls  (WindowPtr theWindow, RgnHandle updateRgn);
pascal void DrawControls    (WindowPtr theWindow);
pascal void Draw1Control    (ControlHandle theControl);
```

Handling Mouse Events in Controls

```
pascal short FindControl    (Point thePoint, WindowPtr theWindow,
                             ControlHandle *theControl);
pascal short TrackControl   (ControlHandle theControl, Point thePoint,
                             ProcPtr actionProc);
pascal short TestControl    (ControlHandle theControl, Point thePt);
```

Changing Control Settings and Display

```
/*some routines have 2 spellings——see Table 5-1 for the alternate spellings*/
pascal void SetControlValue (ControlHandle theControl, short theValue);
pascal void SetControlMinimum
                            (ControlHandle theControl, short minValue);
pascal void SetControlMaximum
                            (ControlHandle theControl, short maxValue);
pascal void SetControlTitle (ControlHandle theControl,
                             ConstStr255Param title);
pascal void HideControl     (ControlHandle theControl)
pascal void MoveControl     (ControlHandle theControl, short h, short v);
pascal void SizeControl     (ControlHandle theControl, short w, short h);
pascal void HiliteControl   (ControlHandle theControl, short hiliteState);
pascal void DragControl     (ControlHandle theControl, Point startPt,
                             const Rect *limitRect,
                             const Rect *slopRect, short axis);
pascal void SetControlAction(ControlHandle theControl, ProcPtr actionProc)
pascal void SetControlColor (ControlHandle theControl,
                             CCTabHandle newColorTable);
```

Determining Control Values

```
/*some routines have 2 spellings——see Table 5-1 for the alternate spellings*/
pascal short GetControlValue
                            (ControlHandle theControl);
pascal short GetControlMinimum
                            (ControlHandle theControl);
pascal short GetControlMaximum
                            (ControlHandle theControl);
pascal void GetControlTitle (ControlHandle theControl, Str255 title);
pascal long GetControlReference
                            (ControlHandle theControl);
pascal void SetControlReference
                            (ControlHandle theControl, long data);
pascal ProcPtr GetControlAction
                             (ControlHandle theControl);
```

```
pascal short GetControlVariant
                            (ControlHandle theControl);
pascal Boolean GetAuxiliaryControlRecord
                            (ControlHandle theControl,
                             AuxCtlHandle *acHndl);
```

Removing Controls

```
pascal void DisposeControl  (ControlHandle theControl);
pascal void KillControls    (WindowPtr theWindow);
```

Application-Defined Routines

Defining Your Own Control Definition Function

```
pascal long MyControl       (short varCode, ControlHandle theControl,
                             short message, long param);
```

Defining Your Own Action Procedures

```
pascal void MyAction        (ControlHandle theControl, short partCode);
pascal void MyIndicatorAction;
```

Assembly-Language Summary

Data Structures

ControlRecord Data Structure

0	`nextControl`	long	handle to next control in control list
4	`contrlOwner`	long	pointer to this control's window
8	`contrlRect`	8 bytes	control's rectangle
16	`contrlVis`	1 byte	value of 255 if control is visible
17	`contrlHilite`	1 byte	highlight state
18	`contrlValue`	word	control's current setting
20	`contrlMin`	word	control's minimum setting
22	`contrlMax`	word	control's maximum setting
24	`contrlDefProc`	long	handle to control definition function
28	`contrlData`	long	data used by control definition function
32	`contrlAction`	long	address of action procedure
36	`contrlRfCon`	long	control's reference value
40	`contrlTitle`	256 bytes	control title (preceded by length byte)

AuxCtlRec Data Structure

0	`acNext`	long	handle to next `AuxCtlRec` record in control list
4	`acOwner`	long	handle to this record's control
8	`acCTable`	long	handle to color table for this control
12	`acFlags`	word	miscellaneous flags
14	`acReserved`	long	reserved for use by Apple Computer, Inc.
18	`acRefCon`	long	for use by application

Global Variables

`AuxCtlHead`	First in a linked list of auxiliary control records
`AuxWinHead`	Contains a pointer to the linked list of auxiliary control records
`DragHook`	Address of procedure to execute during `TrackControl` and `DragControl`
`DragPattern`	Pattern of dragged region's outline (8 bytes)