CHAPTER 8

# Control Panels

8

## Contents

This chapter describes how to develop a control panel to control the settings of systemwide features and how to create an extension for the standard Monitors control panel.

Create a control panel if you want to provide users with the ability to set preferences for global values or systemwide features. Some of the standard control panels allow users to change the speaker volume, set the date and time, and select a different desktop pattern. Although you must not develop control panels to replace the standard ones, you can create additional control panels for any features that meet the stipulations for control panels. If the feature that you want to implement as a control panel is complex or if its interface requires menu items and multiple, layered dialog boxes, you should create a small application instead of a control panel.

If you are a manufacturer of a video device, you can extend the standard Monitors control panel to include items that give users a simple way to control the features of your device. To do this, read the sections in this chapter that describe how to create an extension to the Monitors control panel. The standard Monitors control panel lets the user define the monitor's display of colors and, if more than one monitor is connected to the system, the relative position of each monitor. The Monitors control panel manages any extensions to it that you create.

To use this chapter, you should be familiar with how to create `'BNDL'`, `'ICN#'`, `'FREF'`, signature, and `'DITL'` resources as described in the chapters "Finder Interface" and "Dialog Manager" of *Inside Macintosh: Macintosh Toolbox Essentials*. You should also understand how to handle events and change settings of controls, as explained in the chapters "Event Manager" and "Control Manager" of *Inside Macintosh: Macintosh Toolbox Essentials*.

The Finder, which performs a number of services for your control panel, uses the Dialog Manager to display your control panel's dialog box. In turn, the Dialog Manager uses the Control Manager to create and display buttons, radio buttons, checkboxes, and pop-up menus. Your control panel needs to make these controls active and inactive in response to messages from the Finder. If you include editable text items in your control panel, the Dialog Manager uses TextEdit to handle associated editing tasks. (For general information on TextEdit, see the chapter "TextEdit" in *Inside Macintosh: Text*.)

This chapter provides a general introduction to control panels and introduces the Monitors control panel. It then describes how to

- define the user interface for a control panel
- create the resources for a control panel, including the rectangle and item list resources
- specify the font for your control panel's text
- write a control panel function
- write an extension for the Monitors control panel

# About Control Panels

This section provides an overview of control panels, the resources that a control panel requires, and the Finder's relationship to a control panel. It also distinguishes the services the Finder performs for a control panel from those that your control panel code must implement.

This section also provides an overview of the Monitors control panel and extensions to it, including the resources that a monitors extension requires.

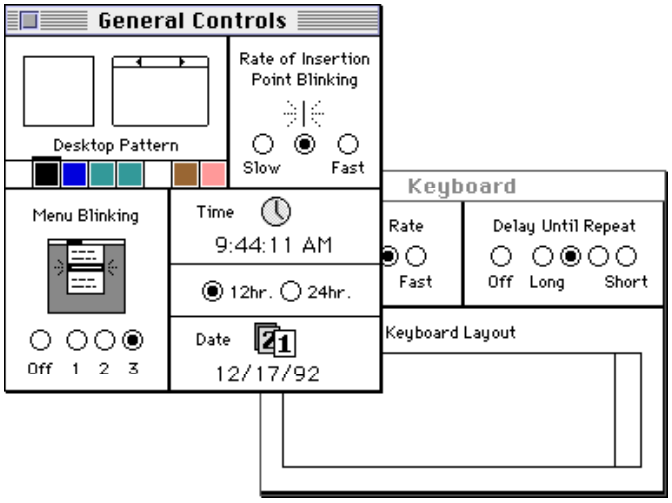
## Control Panels

A **control panel** manages the settings of a systemwide feature, such as the amount of memory allocated to a disk cache, the volume of the speaker, or the picture displayed by a screen saver. A control panel can also allow the user to set a global value, such as the highlight color. On the screen, a control panel appears as a modeless dialog box with controls that let users specify basic settings and preferences for the feature. A **control panel file** (a file of type `'cdev'`) contains the required resources that implement the feature and define the look of the control panel's user interface, including its icon. A control panel file also contains any optional resources needed to implement the feature.

Among the required resources in a control panel file is a code resource that consists of a control device function. A **control device function,** also referred to as a `cdev` function, implements the features of the control panel and performs any services offered by the control panel. Control device functions interact and communicate with the Finder. The Finder provides a number of services for control device functions, including interfacing with the Dialog Manager to create and manage each control panel's dialog box.

A control panel allows the user to modify whatever settings the particular control panel supports. A user opens a control panel from the Finder. Each control panel appears in its own dialog box. Because each control panel is an independent executable file, more than one control panel can remain open at a time, and the user can move among them or run another application while one or more control panels are open. Figure 8-1 shows two control panels open on the desktop. Like other windows, control panels can be dragged on the desktop. The frontmost control panel is the active one.

**Figure 8-1** Two control panels, each with its own window

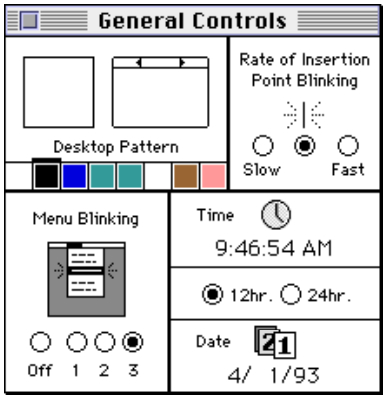



You cannot define your own menus for a control panel, but the user can use most of the Finder's Edit menu commands while working in the control panel. When your control panel is active and the user chooses a command from the Edit menu, the Finder passes the Undo, Cut, Copy, Paste, or Clear commands to your control device function for processing. Your control device function can respond to these messages from the Finder when it is appropriate to do so; for example, if your control panel has an editable text item, your function should respond to editing commands.

Many standard control panels are provided with the system software. For example, the Sound control panel lets the user specify the volume and type of alert sound. The Mouse control panel lets the user define the speed of the onscreen cursor relative to movement of the mouse; the user can also set the double-click speed. The Startup Disk control panel lets the user specify the boot drive.

Figure 8-2 shows the General Controls control panel, which lets the user set the Finder's desktop color and pattern, the blinking rate of the insertion point, the number of times a menu item blinks once the user chooses it, and the time and date.

**Figure 8-2** The General Controls control panel




## A Control Panel's Resources

A control panel file must contain certain required resources. In addition to these, your control panel can include optional resources. You can also create any other types of resources that your control device function needs and include them in the control panel file. The resources you provide in your control panel file must adhere to conventions governing the resource ID numbers; see "Resource IDs for Control Panels" on page 8-14 for information on these conventions. These are the required resources:

- A rectangle positions (`'nrct'`) resource. This resource specifies the number of rectangles that make up the display area of your control panel and a list of the coordinates defining the position for each rectangle. (Your control panel interface can have one or more rectangles containing the controls that let the user set and change values or otherwise manipulate the feature the control panel governs.)
- An item list (`'DITL'`) resource. This resource specifies the items in your control panel. You can specify in this resource items such as static text, buttons, checkboxes, radio buttons, editable text, user items, icons, QuickDraw pictures, and other types of controls, such as pop-up menus.
- A machine (`'mach'`) resource. This resource specifies the types of systems on which your control panel can run.
- A black-and-white icon list (`'ICN#'`) resource and other resources associated with an icon family. These resources define the icon for your control panel file. The icon family resources are `'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to create an icon family.

- A bundle (`'BNDL'`) resource. This resource groups together the control panel's signature, icon, and file reference resources.
- A file reference (`'FREF'`) resource. This resource associates icons with your control panel file; the Finder uses this information to display the icon for your control panel file.
- A signature resource. This resource contains a unique four-character sequence that has the same value as your control panel's creator type.
- A control device (`'cdev'`) code resource. This resource contains the code that implements the control panel.

Although it is not required, you can also include a font information (`'finf'`) resource in your control panel file. This resource type lets you specify the font of your control panel's static text items. If you don't include a font information resource, the Finder uses the default application font, which is 9-point Geneva for Roman scripts.

The control device code resource contains a control device function, which must be the first section of code in the resource. The control device function handles messages from the Finder and implements the work your control panel is designed to do. The Finder handles such actions as displaying your control panel's dialog box and tracking controls in it.

## The Finder's Interaction With Control Panels

The Finder performs the following services on behalf of your control panel:

- queries your control device function initially, to determine whether it can run on the available software and hardware configuration
- requests your control device function to perform any needed initialization when the user first opens your control panel
- displays dialog items defined by your control panel file
- tracks user actions in controls defined by your control panel file
- manages the modeless dialog box in which your control panel is displayed (For instance, the Finder responds appropriately when the user drags the modeless dialog box or clicks its close box.)
- sends your control device function the information it needs to respond to specific events or to handle Edit menu commands
- displays messages to the user when the control panel cannot run on the current system and when your control device function returns an error code

Your control panel should

- provide both the required resources and any additional resources that the Finder needs to run your control panel
- initialize, open, and close your control panel appropriately as requested by the Finder

- respond to activate events as requested by the Finder
- draw user items in response to update events as requested by the Finder
- respond to user actions in controls as requested by the Finder
- respond to user keystrokes as requested by the Finder

## Control Panels and System Extensions

Many control panels rely on system extensions (files of type `'INIT'`) to implement their features. For example, you might implement a screen saver as a system extension and create a control panel that allows the user to set specific features of the screen saver, such as the color of the picture displayed. Although the extension creates and manages the screen saver, the user might control the look of the screen saver through settings in the control panel. In this scenario, which is used as an example throughout this chapter, the control device function and its system extension communicate and share values related to settings that the user changes.

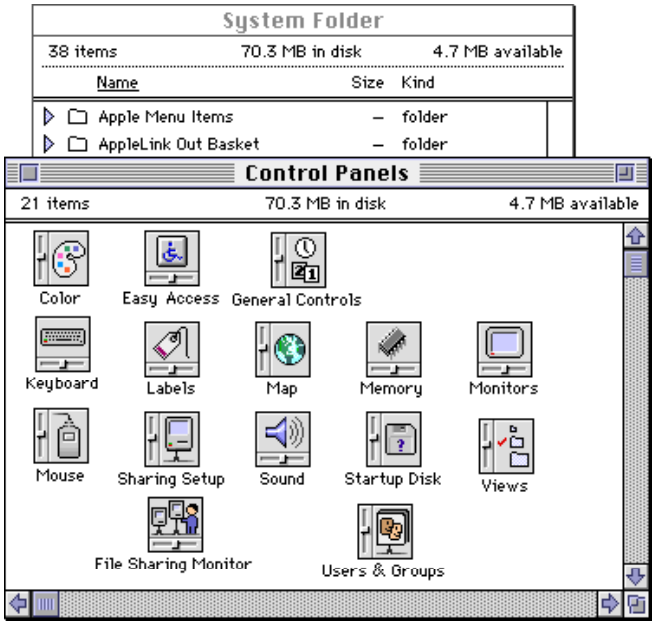
If you use a system extension with your control panel, include it in the control panel file along with the required resources and any other optional resources you use. In System 7, system extensions can be installed in the Control Panels folder or the Extensions folder (both of which are stored in the System Folder) or directly in the System Folder. However, if it contains a system extension, your control panel file must reside in the Control Panels folder within the System Folder. At startup time, the system software opens files of type `'cdev'` that reside in the Control Panels folder and executes any system extensions that it finds there. If the system extension portion of a control panel is not loaded at startup, the control panel won't function properly.

## About User Documentation for Control Panels

Because control panels are like independent files, you or the user can install and store them anywhere in the file system. Users might want to store frequently used control panels in the Apple Menu Items folder or in a folder containing other utilities.

You should refer to a file of type `'cdev'` as a *control panel file* in any user documentation that you provide. Don't refer by name to the file type of this file or any other file. If your control panel file includes a system extension, you should direct the user to install it in the Control Panels folder or provide an installation script for this purpose. System software provides an alias (a file that points to another file) of the Control Panels folder for quick access from the Apple menu. Figure 8-3 shows many control panel icons in the Control Panels folder.

**Figure 8-3** Control panel icons in the Control Panels folder




## The Monitors Control Panel and Extensions to It

The standard Monitors control panel lets the user define the monitor's display of colors or shades of gray. If more than one monitor is connected to the system, the Monitors control panel also allows the user to define the relative position of each monitor and choose which monitor is the startup screen. If you are a manufacturer of a video card, you can create a monitors extension to give users a simple way to control the features of your device through the Monitors control panel. A monitors extension controls the features of your video card only, not systemwide features. For example, a monitors extension might allow the user to set the virtual screen size for a single monitor but not the size of the menu bar, which can appear on any monitor. If you require a more complex interface, such as your own menu items or several levels of nested dialog boxes, you should create a small application rather than an extension to the Monitors control panel.

The Monitors control panel manages any extensions to it that you create, and the user can open an extension only through the Monitors control panel. Like a control panel file, a monitors extension file has a file type of `'cdev'`. A monitors extension file contains resources for the monitors extension, including a code resource of type `'mntr'`. If you want to create a separate control panel to let the user control the settings of another feature of the same video card, you can include the control panel's resources and code in the same file as your monitors extension. In this case, you create the control panel just as you do any other independent control panel. If a user opens your independent control

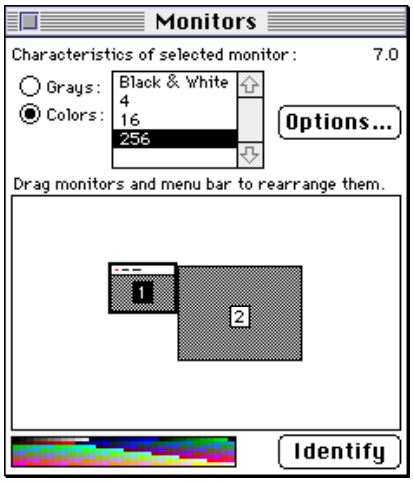
panel, the Finder displays the control panel defined in your file and ignores the monitors extension in that file, just as the Monitors control panel ignores the independent control panel defined in your file when it opens the file to display the monitors extension.

The Monitors control panel allows a user to

- select which one of the monitors connected to the computer to use as a startup screen (that is, which monitor displays the menu bar)
- inform system software about the relative locations of the monitors
- control some features of the monitors, for instance, how many colors or shades of gray are displayed

Figure 8-4 shows an example of the Monitors control panel.

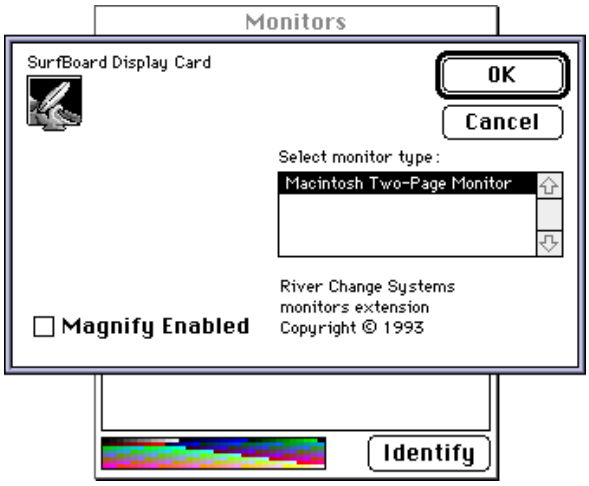
**Figure 8-4** The Monitors control panel




If more than one video card is installed in the computer, the Monitors control panel shows all of the connected monitors. When the user selects one monitor, then clicks the Options button, the Monitors control panel displays the Options dialog box for that monitor. When you provide a monitors extension for the Monitors control panel, the controls you add appear in this dialog box.

Figure 8-5 shows an example of an Options dialog box for the SurfBoard video card. The OK and Cancel buttons are standard for all Options dialog boxes. In this example, the developers of the SurfBoard video card have provided a monitors extension that adds two items to the the Options dialog box: the Magnify Enabled checkbox and static text listing the manufacturer's name.

**Figure 8-5** An Options dialog box for the SurfBoard video card






A monitors extension file must contain these four resources:

- A card (`'card'`) resource. This resource contains a Pascal string identical to the name stored in the declaration ROM of the video card. You can include as many card resources as you like, so that one extension file can handle several types of video cards.
- A monitor (`'mntr'`) code resource. This resource carries out the functions of your monitors extension.
- A rectangle (`'RECT'`) resource. This resource describes the size and shape of the area that your controls occupy.
- An item list (`'DITL'`) resource specifying the items in your monitors extension. You can also add additional controls, separated from other controls by a horizontal line, for the benefit of advanced users (superusers).

Your monitors extension file can also include any of the following resources:

- One or more members of an icon family (`'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`), each with resource ID –4064, that define an icon for your video card. If you provide any of these resources, the Monitors control panel displays the appropriate icon from the icon family in the upper-left corner of the Options dialog box.
- Additional icon family resources to provide a unique icon for your monitors extension file.
- A version (`'vers'`) resource. This resource provides version information for your monitors extension.
- A string list (`'STR#'`) resource defining one or more video card names. If you want the Options dialog box to display a name that is different from the one in the declaration ROM of the card, define the alternate name in an `'STR#'` resource.

- One or more gamma table (`'gama'`) resources. Here you can include gamma tables that allow your video card to provide the most accurate colors possible.
- A file reference (`'FREF'`) resource. This resource associates icons with your monitors extension file; the Finder uses this information to display the icon for your monitors extension file.
- A bundle (`'BNDL'`) resource. This resource groups together the monitor extensions' signature, icon, and file reference resources.
- A system extension (`'INIT'`) resource. Although this resource acts independently of other resources in the file, it should be related to the monitors extension.
- A signature resource (of type `'STR '`).

# Creating Control Panel Files

This section describes how to create your control panel's resources, including the code resource that implements the control panel. This section discusses how to

- define the user interface for your control panel
- create resources for your control panel, including those that define
  - control panel rectangles
  - the item list resource
  - icons
  - the machine resource
  - the file reference, bundle, and signature resources
- specify the font of static text in your control panel
- write a control panel function

Before you begin, consider whether the feature that you have in mind is best governed by a control panel. It should be a systemwide feature amenable to manipulation by the user, who would use the control panel only occasionally to set or change preferences. If you find that you need special menus or nested dialog boxes to implement your control panel, create a small application instead.

## Defining the User Interface for a Control Panel

The user interface for a control panel consists of the display area defined by the dialog box and its controls, including checkboxes, buttons, static text, editable text, and user items. In addition, you need to provide an icon for your control panel file, for display by the Finder. A control panel can open in a modeless dialog box of any size, limited only by the screen display.

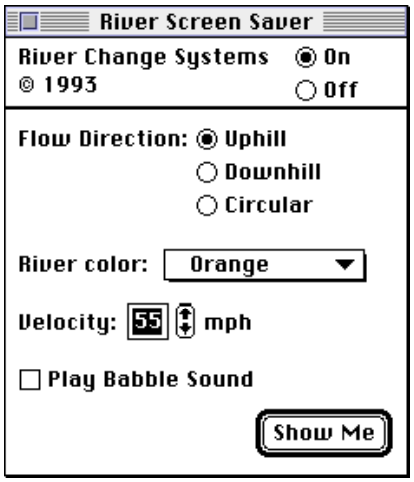
Your control panel's display area can consist of one or more rectangles; you determine the display area by defining the rectangles and their positions. You specify these values in your control panel's rectangle positions (`'nrct'`) resource. These rectangles essentially determine the size of the dialog box. The Finder calculates the boundaries of the dialog box from the coordinate values you specify in your rectangle positions resource.

When deciding on the size and number of your rectangles, consider the number and placement of the buttons, checkboxes, text, and other items in your control panel. Allow enough space for the user to distinguish them easily. Because control panels are generally used only occasionally, make the interface as simple as possible. If you choose the default settings well, the user should seldom need to use your control panel.

Figure 8-6 shows the user interface for the River control panel used as an example throughout this chapter. It governs certain features of River, a screen saver system extension.



**Figure 8-6** The River control panel interface




In System 7, you can include a font information resource that specifies the font in which the Finder displays your control panel's static text items. (For information on creating a font information resource, see "Specifying the Font of Text in a Control Panel" on page 8-23.) Choose a font that is easy to read. In System 7, the control panel interface allows ample space for larger point sizes; Apple recommends 12-point Chicago.

If you don't include a font information resource, the Finder uses the default application font for static text items. For Roman scripts, this is 9-point Geneva. (The static text of the River control panel illustrated in Figure 8-6 is 12-point Chicago because this control panel provides a font information resource for this purpose.) Note that the Finder uses the system font to draw text strings that you define as part of a control item in your item list; for Roman scripts, this is 12-point Chicago.

If your control panel runs in both System 7 and System 6 but you wish to display your control panel's static text in 12-point Chicago, you can define the text as user items. See "Defining Text in a Control Panel as User Items" on page 8-24 for details.

If you wish, you can create an icon family to specify the icon that the Finder uses to represent your control panel file. The icon family resources are `'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`.

The icons for a control panel file are square and include a horizontal or vertical slider along with a graphic representing the feature governed by the control panel. Figure 8-7 shows an icon for the River control panel file.

**Figure 8-7** An icon for the River control panel file

See *Macintosh Human Interface Guidelines* for more information on designing an icon. For complete information on designing a dialog box, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

## Creating a Control Panel's Resources

The following sections describe the required and optional resources that you supply for your control panel. The first section contains general information that applies to all of the individual resources. Later sections discuss each of the required resources and some optional resources.

### Resource IDs for Control Panels

Every resource has a resource ID. With one exception, all resource IDs for control panel resources, including standard resources and resources you define yourself, must be in the range of –4064 through –4033. The exception is the resource for the icon help balloon (`'hfdr'`) resource, whose resource ID is –5696.

Of this range, resource IDs from –4064 through –4049 are reserved for standard resources and some optional resources.

You can assign resource IDs in the range –4048 through –4033 to any private resources that you define for your control panel.

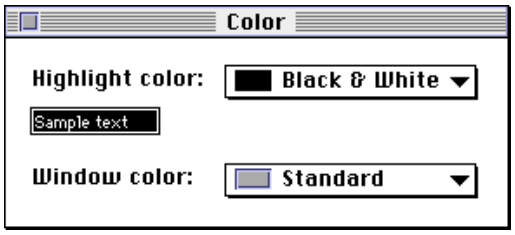
**Note**
You can use a high-level tool such as the ResEdit application, which is available through APDA, to create your resources. (See *ResEdit Reference* for details on using ResEdit.) You can also use the Rez utility. ◆

## Defining the Control Panel Rectangles

Your control panel can consist of one rectangle, as in Figure 8-8, or several (see Figure 8-2 on page 8-6 and Figure 8-6 on page 8-13). You define these rectangles in a rectangle positions (`'nrct'`) resource. You specify in this resource the number of rectangles for your control panel and a list of the coordinates for each rectangle. You must specify a resource ID of –4064 for a rectangle positions resource.



**Figure 8-8** The Color control panel




In the rectangle positions resource you specify a rectangle's coordinates in this order: top, left, bottom, and right. Although you can define a control panel of any size (limited only by the screen display), you must specify the coordinates (–1,87) as the origin (upper-left point) of the upper-left rectangle. To provide for backward compatibility with the Control Panel desk accessory, the Finder accepts only these coordinates as the origin of a control panel. If you are designing for System 7 only, you can extend the bottom and right edges of a control panel as far as you like. If you want your control panel to run in System 7 *and* previous versions of system software, you must limit your control panel's size to the area bounded by (–1,87,255,322). These are the coordinates used by the Control Panel desk accessory.

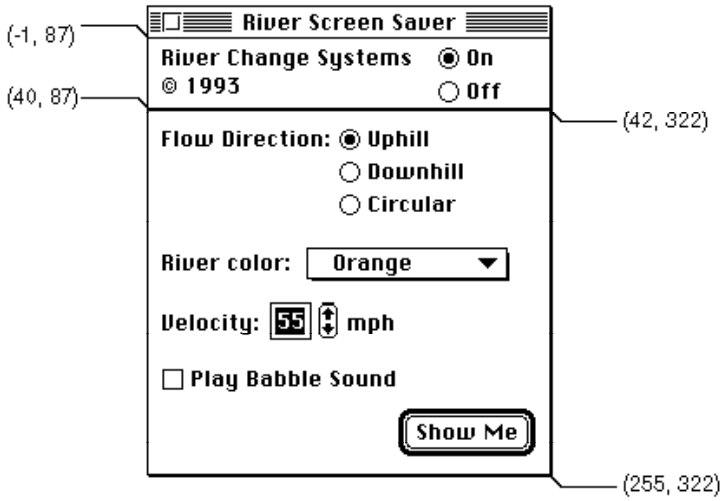
#### The Control Panel Desk Accessory

In System 6, the Control Panel desk accessory is a single interface shared by all control panels. It has two parts: a scrollable list of icons representing the control panels a user can open as part of the desk accessory and a display area of fixed size. If you want to make your control panel compatible with the Control Panel desk accessory, it must fit in this area. The Control Panel desk accessory acts as a driver interfacing with and managing the control panels whose icons it displays. All of the control panels represented by icons in the scrollable list share the same display area. For this reason, a user can open only one control panel at a time. ◆

If you want to make your control panel backward compatible, remember that the Control Panel desk accessory draws a frame that is 2 pixels wide around each rectangle. To join two parts of a panel neatly, overlap their rectangles by 2 pixels on the side where they meet.

Figure 8-9 shows the coordinates of the two rectangles that make up the River control panel. Because the River control panel has relatively few items, they fit well within the space constraints imposed by the Control Panel desk accessory. Thus, this control panel can run in both the Finder in System 7 and the Control Panel desk accessory in System 6.

**Figure 8-9** Coordinates defining the rectangles of the River control panel display area




Listing 8-1 shows the Rez input for the rectangle positions resource that specifies the rectangles for the River control panel.

**Listing 8-1** Rez input for a rectangle positions list (`'nrct'`) resource

```
resource 'nrct' (-4064, purgeable) {
 { /*array RectArray: 2 elements*/
   /*[1]*/
   {-1, 87, 42, 322},
   /*[2]*/
   {40, 87, 255, 322}
 }
};
```

If you define two or more rectangles that together do not form a complete square or rectangle in relation to the bounding dialog box that the Finder creates, the Finder fills in any blank space on the control panel with a gray pattern.

**Note**
In System 6, the Control Panel desk accessory first fills in the area defined by the coordinates (–1,87,255,322) with a gray background pattern. It then creates white areas corresponding to the rectangles you define. In these, it draws the items of your control panel. The Control Panel desk accessory outlines the rectangles with a 2-pixel-wide frame. ◆



## Creating the Item List Resource

You define the items in your control panel and their positions within its rectangles using an item list (`'DITL'`) resource. These items can include static text, buttons, checkboxes, radio buttons, editable text, the resource IDs of icons and QuickDraw pictures, and the resource IDs of other types of controls, such as pop-up menus. You must specify a resource ID of –4064 for your control panel's item list resource.

An item list contains a display rectangle for each item. A display rectangle determines the size and location of the item. You must specify the coordinates of an item's display rectangle relative to the origin of the control panel's upper-left rectangle.

Recall that the origin (the point at the extreme upper left) of your control panel must coincide with the coordinates (–1,87). In the Control Panel desk accessory, the origin is at the upper left of the rectangle containing the scrollable list of icons, to the left of the display area. A 2-pixel-wide frame borders the rectangle containing the scrollable list of icons.

Listing 8-2 shows the item list resource for the River control panel. Notice that the item list includes a static text item (item 2) giving the control panel's name and copyright. The upper-left point of the display rectangle for the static text lies at the coordinates (4,95).

In Listing 8-2, some items are defined as enabled and some as disabled. By specifying each item in the item list as enabled or disabled, you inform the Dialog Manager whether or not to report user clicks in the item.

Depending on the type of item, you usually provide a text string or a resource ID for the item.

Note that text in a control panel is defined either as part of a control (such as labels for buttons, checkboxes, radio buttons, and pop-up menus), or as separate items (static text, editable text, or user items). For example, the text "River color" is defined as part of a pop-up control in a separate menu resource and the text "mph" is defined as a static text item.

The item list resource for the River control panel defines text that is not provided by a control as static text items; in addition to the product name, these static text items include "Flow Direction:" and "Velocity:" (see Figure 8-6 on page 8-13). The item list resource defines one editable text item, setting the default text for this item to 55. It also defines the editable text item as disabled. If you define an editable text item as disabled, the Dialog Manager and TextEdit handle user input in the editable text item.

**IMPORTANT**

If you want to use a font other than the default application font for your control panel's text *and* you want your control panel to run in the Control Panel desk accessory of System 6, you must define the text as user items instead of static text items. For more information on this, see "Defining Text in a Control Panel as User Items" on page 8-24. ◆

In Listing 8-2, the first item in this resource is an enabled button labeled "Show Me." This is the River control panel's default button. (The Control Manager positions the label inside the button and draws it using the system font.) Notice that the outline around the button, which identifies it as the default button, is defined as a separate item (a disabled user item) toward the end of the listing.

All of the other controls with which the user interacts are defined as enabled—the On and Off radio buttons, the radio buttons beside the label Flow Direction, the Play Babble Sound checkbox, and the River color pop-up control. When these controls are active, the user can click them, changing settings and making selections. The up and down arrows are defined as enabled user items, and the item list resource includes a picture item that refers to a resource containing a QuickDraw picture of the arrows. Finally, the item list resource includes a help item referencing the resource ID that defines the help balloons for the River control panel.

**Listing 8-2** Rez input for an item list (`'DITL'`) resource

```
resource 'DITL' (rControlPanelDialog, purgeable) {
   { /*array: 18 elements*/
   /*[1]*/
   {219, 237, 239, 308},
   Button {       enabled,       "Show Me"                      },
   /*[2]*/
   {4, 95, 44, 247},
   StaticText {   disabled,      "River Change Systems\n© 1993" },
   /*[3]*/
   {2, 254, 21, 302},
   RadioButton {  enabled,       "On"                           },
   /*[4]*/
   {22, 254, 40, 302},
   RadioButton {  enabled,       "Off"                          },
   /*[5]*/
   {51, 95, 70, 196},
   StaticText {   disabled,      "Flow Direction:"              },

   /*[6]*/
   {50, 197, 68, 303},
   RadioButton {  enabled,       "Uphill"                       },
```

```
   /*[7]*/
   {69, 197, 87, 303},
   RadioButton {  enabled,       "Downhill"                    },
   /*[8]*/
   {88, 197, 106, 303},
   RadioButton {  enabled,       "Circular"                    },
   /*[9]*/
   {157, 95, 178, 156},
   StaticText {   disabled,      "Velocity:"                   },
   /*[10]*/
   {156, 162, 172, 180},
   EditText {     disabled,      "55"                          },
   /*[11] (up arrow)*/
   {150, 184, 162, 201},
   UserItem {     enabled,                                     },
   /*[12] (down arrow)*/
   {163, 184, 175, 201},
   UserItem {     enabled,                                     },
   /*[13] (picture of up/down arrows)*/
   {150, 184, 175, 201},
   Picture {      disabled,      –4048                         },
   /*[14]*/
   {157, 202, 176, 242},
   StaticText {   disabled,      "mph"                         },
   /*[15] (outline around default button)*/
   {212, 231, 247, 314},
   UserItem {     disabled,                                    },
   /*[16]*/
   {188, 95, 208, 241},
   Checkbox{      enabled,       "Play Babble Sound"           },
   /*[17] (title & menu items defined by menu w/res ID mPopUp)*/
   {122, 92, 142, 297},
   Control {      enabled,       mPopUp                        },
   /*[18] get help balloon information from 'hdlg' resource*/
   {0,0,0,0},
   HelpItem {     disabled,
            HMScanhdlg  /*scan resource type—'hdlg' or 'hrct'*/
            {–4064}                                           }
   }
};
```

For complete information on creating an item list resource, see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

## Defining the Icon for a Control Panel

You create an icon family to specify the icon that the Finder uses to represent your control panel file. The icon family resources are `'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`. You must specify a resource ID of –4064 for the icon family resources of a control panel and mark these resources as purgeable. If you provide the complete icon family, the Finder displays the appropriate icon family member according to the bit depth of the monitor. For more information on these icons, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

## Specifying the Machine Resource

When the user opens your control panel, the Finder reads your machine (`'mach'`) resource from your control panel file. Depending on the value you specify in the machine resource, the Finder takes one of two actions: (1) calls your control device function, directing your function to check the current hardware and software configuration to determine whether your control panel can run on the current system; or (2) performs the check itself. You must specify a resource ID of –4064 for a machine resource.

The machine resource consists of a hard mask and a soft mask. The Finder handles the check if you set these masks to values indicating that your control panel runs on all systems or to values representing the requirements for your control panel; the Finder checks the current configuration in the latter case. If the Finder handles the check, it never calls your control device function with a `macDev` message; instead, the Finder calls your function for the first time with an initialization message. If the Finder determines that your control panel cannot run on the current system, the Finder displays an alert box to the user and does not open the control panel. (In System 6, the Control Panel does not display the icon for a control panel file if the machine resource indicates the control panel cannot run on the current system.)

If you set the hard mask to $FFFF and the soft mask to $0000, indicating your control device function performs its own requirements check, the Finder calls your function with a `macDev` message once only, and this is the first call the Finder makes to your function. (See"Determining If a Control Panel Can Run on the Current System" on page 8-29 for a discussion of how to handle a `macDev` message.)

Table 8-1 shows the values you use to set the machine resource masks.

**Table 8-1** Possible settings for the machine resource masks

| Soft mask | Hard mask | Action |
|---|---|---|
| $0000 | $FFFF | The Finder calls this control device function with a `macDev` message, and the function must perform its own hardware and software requirements check. |
| $3FFF | $0000 | This control panel runs on Macintosh II systems only. |
| $7FFF | $0400 | This control panel runs on all systems with an Apple Desktop Bus (ADB). |
| $FFFF | $0000 | This control panel runs on all systems. |



Listing 8-3 shows the Rez input for a machine resource. The values in this machine resource indicate to the Finder that the control panel performs its own hardware and software requirements check.

**Listing 8-3** Rez input for a machine (`'mach'`) resource

```
resource 'mach' (-4064, purgeable) {
   0xFFFF,           /*hard mask*/
   0                 /*soft mask*/
};
```

**Note**

The machine resource allows the Finder to cache information about each control panel. The user can force the Finder to rebuild the cache by pressing Command-Option while opening the control panel. ◆

## Creating the File Reference, Bundle, and Signature Resources

You must create a file reference resource, a signature resource, and a bundle resource to enable the Finder to display the icon for your control panel. You must specify a resource ID of –4064 for both a bundle resource and a file reference resource.

The file reference resource specifies a file type (for a control panel, `'cdev'`), the local resource ID of an icon list resource, and an empty string. The local ID maps the file type (`'cdev'`) to your icon list resource that is assigned the same local ID in the bundle resource. Listing 8-4 shows the file reference resource for the River control panel.

**Listing 8-4** Rez input for a file reference (`'FREF'`) resource

```
resource 'FREF' (-4064, purgeable) {
   'cdev', 0, ""
};
```

The Finder uses the signature resource with the bundle resource to establish your control panel's identity. You define a signature resource as a string resource (that is, a resource of type `'STR '`) and specify as its resource type a unique four-character sequence that has the same value as your control panel's creator type. A signature resource has a resource ID of 0.

The signature resource contains a string that identifies your control panel; typically the string specifies the name, version number, and release date of your control panel. Listing 8-5 shows the River control panel's signature resource, which has a signature of `'rivr'`, in Rez input format.

**Listing 8-5** Rez input for a signature resource

```
type 'rivr' as 'STR ';
resource 'rivr' (0, purgeable) {
   "River Control Panel 1.0"
};
```

A bundle (`'BNDL'`) resource associates all of the resources that the Finder uses for your control panel. It associates your control panel file and your control panel's signature with its icon. The Finder requires the information in the bundle resource in order to display icons for your control panel. In the bundle resource, you must assign a local ID to your icon list resource that matches the local ID you assigned inside the corresponding file reference resource. In the bundle resource shown in Listing 8-6, local ID 0 is assigned to the icon list resource with a resource ID –4064, which maps the icon defined for the River control panel to the control panel file.

**Listing 8-6** Rez input for a bundle (`'BNDL'`) resource

```
resource 'BNDL' (-4064, purgeable) {
   'rivr', 0,
   {  'ICN#', {0, -4064},
      'FREF', {0, -4064}
   }
};
```

(See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information on how to create file reference, signature, and bundle resources.)

## Providing Additional Resources for a Control Panel

In addition to providing required resources, you can supply optional resources for your control panel. For example, you can supply resources to store the settings of controls, text strings, or font information. The River control panel stores its controls' settings in a resource that it defines.

If you wish, you can provide help balloon resources. For example, you can include a resource to define a help balloon for your control panel's icon in the Finder. The resource type of an icon help balloon resource is `'hfdr'`, and its resource ID is –5696. This is the only control panel resource whose resource ID is outside the range of –4064 through –4033.

You can also include help balloon resources for specific items or areas of your control panel. For example, you might include a help balloon resource to explain how to use a control. For this purpose, you supply a resource of type `'hdlg'` or `'hrct'` with a resource ID of –4064. For information on how to create help balloon resources, see the chapter "Help Manager" in this book.

If you define any other types of resources for your control panel, you must assign them resource IDs in the range –4048 through –4033.



## Specifying the Font of Text in a Control Panel

A control panel typically contains uneditable text that is part of a control item or defined as static text. See Listing 8-7 on page 8-24 for examples.

The Finder uses the default application font to draw control panel items that you define as static text. However, you can specify that a different font be used for this purpose. There are two ways to do so. The easiest way is to define a font information (`'finf'`) resource. This is the method you should use if you intend your control panel to run in System 7 only.

If you want your control panel to be compatible with the Control Panel desk accessory, you cannot use this method because the Control Panel desk accessory does not recognize font information resources. In this case, you can use an alternative method, which entails defining your control panel's static text as user items, setting the font, and drawing the text. This section explains both methods.

You can also specify the font to be used for each item by creating an item color table (`'ictb'`) resource whose entries correspond to the items in your item list. However, you cannot use this method in System 6, because the Control Panel desk accessory appends your control panel's item list to its own. For more information about the item color table (`'ictb'`) resource, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

### Creating a Font Information Resource

You create a font information (`'finf'`) resource to specify the font in which the Finder displays your control panel's static text. You include the font information resource in your control panel file, and the Finder reads this resource when it opens your file. You must use the resource ID –4049 for a font information resource.

In the font information resource, you specify the font ID number, the font style, and its size. The Finder sets the graphics port's `txFont`, `txFace`, and `txSize` fields to the values you specify, and QuickDraw draws the text using these values.

## Defining Text in a Control Panel as User Items

If you want to specify the font for your control panel's text and also want your control panel to run in both System 6 and System 7, you can define your control panel's text as disabled user items rather than as disabled static text items. Your control device function must call QuickDraw to set the graphics port fields for the font and its characteristics, and then draw the text at initialization and in response to update events. See "Handling Text Defined as User Items" on page 8-43 for more information.

For these user items, you can define a string list (`'STR#'`) resource to store the text strings that make up your text. Your control device function can read the text from the string list resource and store the text in a data structure in your control device function's private storage. If you do this, then your control device function can read the values from its private storage whenever it needs to update user items.

Listing 8-7 shows a part of the River control panel's item list with the control panel's text defined as user items. Because the user does not need to read the product title and copyright regularly to interact with the control panel, the control panel defines this string as a static text item; the Finder draws this text string only in 9-point Geneva. The control panel defines all other text strings as user items, and the control device function sets the font and draws those user items containing text.

**Listing 8-7** A control panel's static text defined as user items

```
resource 'DITL' (rControlPanelDialog, purgeable) {
   { /* array DITLarray: 18 elements */ }
   /* . . . */
   /* [2] */
   {4,95,44,247},
   StaticText {   disabled,      "River Change Systems\n© 1993"
},
   /* . . . */
   /* [5] */
   {51, 95, 70, 196},
   UserItem {     disabled,      /*Flow Direction:*/            },

   /* . . . */
   /* [9] */
   {157, 95, 178, 156},
   UserItem {     disabled,      /*Velocity:*/                  },
   /* . . . */
```

```
   /* [14] */
   {157, 202, 176, 242},
   UserItem {     disabled,      /*mph*/                    },
   /* . . . */
   }
};
```



## Writing a Control Panel Function

A control panel requires a control device (`'cdev'`) code resource, which contains the code that implements the feature your control panel provides. The first piece of code in this resource must be a control device function that adheres to a defined interface. When the user opens your control panel, the Finder loads your code resource (of type `'cdev'`) into memory.

The Finder calls your control device function, requesting it to perform the action indicated by the `message` parameter, in response to events and the user's interaction with your control panel. Your control device function should perform the requested action and return a function result to the Finder. Your control device function should return as its function result either a standard value indicating that it has not allocated storage, a handle to any storage it has allocated, or an error code. Here is how you declare a control device function:

```
FUNCTION MyCdev(message, item, numItems, CPrivateValue: Integer;
                VAR theEvent: EventRecord;
                cdevStorageValue: LongInt;
                CPDialog: DialogPtr): LongInt;
```

The `message` parameter can contain any of the values defined by these constants:

```
CONST
   macDev      = 8;  {determine whether control panel can run}
   initDev     = 0;  {perform initialization}
   hitDev      = 1;  {handle click in enabled item}
   updateDev   = 4;  {respond to update event}
   activDev    = 5;  {respond to activate event}
   deActivDev  = 6;  {respond to control panel becoming inactive}
   keyEvtDev   = 7;  {respond to key-down or auto-key event}
   undoDev     = 9;  {handle Undo command}
   cutDev      = 10; {handle Cut command}
   copyDev     = 11; {handle Copy command}
   pasteDev    = 12; {handle Paste command}
   clearDev    = 13; {handle Clear command}
   nulDev      = 3;  {respond to null event}
   closeDev    = 2;  {respond to user closing control panel}
```

These constants (as specified in the `message` parameter) indicate that your control device function should perform the following actions:

- `macDev`. Determine whether the control panel can run on the current system, and return a function result of 1 if it can and 0 if it cannot.
- `initDev`. Perform initialization.
- `hitDev`. Handle a click in an enabled item.
- `updateDev`. Update any user items and redraw any controls that are not standard dialog items handled by the Dialog Manager.
- `activDev`. Respond to your control panel becoming active by making the default button and any other controls in your control panel active.
- `deActivDev`. Respond to your control panel becoming inactive by making the default button and any other controls in your control panel inactive.
- `keyEvtDev`. Handle a key-down or auto-key event.
- `undoDev`. Handle an Undo command.
- `cutDev`. Handle a Cut command.
- `copyDev`. Handle a Copy command.
- `pasteDev`. Handle a Paste command.
- `clearDev`. Handle the Clear command.
- `nulDev`. Handle a null event by performing any idle processing.
- `closeDev`. Handle a click in the close box by terminating, after disposing of any handles and pointers created by your function.

The control device function that implements the River control panel used as an example in this chapter shows one way of handling messages from the Finder. In this scenario, the user sets the screen saver's characteristics using the River control panel. The River control panel (`'cdev'`) file includes a system extension that displays the screen saver when the user signals it to do so. The River control panel uses the system extension to display the screen saver using the current settings whenever the user clicks the panel's default button (Show Me). (See Figure 8-6 on page 8-13.)

The River control device function reads control settings from a resource stored in its preferences file, which is stored in the Preferences folder, and writes new values to that file at certain points after the user changes control settings. The control device function alerts the system extension of changes in the preferences file, and the system extension gets the new values to use from the preferences file.

In addition to the required resources, the River control device function uses a number of private resources that are included in the control panel file.

Listing 8-8 shows the River control panel's control device function, called `main`. To respond to requests from the Finder, the function uses a CASE statement that handles each type of message sent by the Finder.

The remainder of this section discusses each of these messages in detail and includes code showing how the River control panel processes the messages.

**Listing 8-8** A control device function

```
UNIT RiverCP;
INTERFACE
      {include a Uses statement if your programming environment requires it}
CONST
   kShowMe                 = 1;
   kOnRadButton            = 3;
   kOffRadButton           = 4;
   kUphillRadButton        = 6;
   kDownhillRadButton      = 7;
   kCircularRadButton      = 8;
   kVelocityEditText       = 10;
   kUserItemUpArrow        = 11;
   kUserItemDownArrow      = 12;
   kPict                   = 13;
   kUserItemButtonOutline  = 15;
   kBabbleCheckBox         = 16;
   kRiverColorMenu         = 17;
TYPE
   MyRiverStorage =
      RECORD
         err:              LongInt;
         count:            LongInt;
         settingsChanged:  Boolean;
      END;
   MyRiverStoragePtr    = ^MyRiverStorage;
   MyRiverStorageHndl   = ^MyRiverStoragePtr;

FUNCTION main (message, item, numItems, CPrivateValue: Integer;
               VAR theEvent: EventRecord; cdevStorageValue: LongInt;
               CPDialog: DialogPtr): LongInt;

IMPLEMENTATION
FUNCTION main;

{any support routines used by your control panel function}

VAR
   myRiverHndl:         MyRiverStorageHndl;
   initDevOrMacDevMsg:  Boolean;
   okToRun:             LongInt;
   cpMemError:          Boolean;
BEGIN
   cpMemError := MyRoomToRun(message, cdevStorageValue);
```

```
   IF cpMemError THEN      {an error occurred or there isn't enough memory }
      main := cdevMemErr   { to run, return immediately}
   ELSE {handle the message}
   BEGIN
      IF (message <> macDev) AND (message <> initDev) THEN
         myRiverHndl := MyRiverStorageHndl(cdevStorageValue);
      CASE message OF
         macDev: {check machine characteristics}
            BEGIN
               MyCheckMachineCharacteristics(okToRun);
               main := okToRun;
            END;
         initDev:{perform initialization}
            MyInitializeCP(cdevStorageValue, CPDialog, myRiverHndl);
         hitDev: {user clicked dialog item}
            BEGIN
               item := item - numItems;
               MyHandleHitInDialogItem(item, cdevStorageValue,
                                       CPDialog, myRiverHndl);
            END;
         activDev: {control panel is becoming active}
            MyActivateControlPanel(cdevStorageValue, CPDialog,
                                   myRiverHndl, TRUE);
         deActivDev: {control panel is becoming inactive}
            MyActivateControlPanel(cdevStorageValue, CPDialog,
                                   myRiverHndl, FALSE);
         updateDev: {update event -- draw any user items}
            MyUpdateControlPanel(cdevStorageValue, CPDialog, myRiverHndl);
         cutDev, copyDev, pasteDev, clearDev: {editing command}
            MyHandleEditCommand(message, CPDialog);
         keyEvtDev: {keyboard-related event}
            MyHandleKeyEvent(theEvent, CPDialog, message);
         nulDev:  {null event -- perform idle processing}
            MyHandleIdleProcessing(cdevStorageValue, CPDialog, myRiverHndl);
         closeDev: {user closed control panel, release memory before exiting}
            MyCloseControlPanel(myRiverHndl, cdevStorageValue);
      END; {of CASE}
      IF message <> macDev THEN
         main := LongInt(cdevStorageValue);
   END;     {of handle message}
END;  {of main program}
END.
```

When the Finder first calls your control device function, the current resource file is set to your control panel (`'cdev'`) file, the current graphics port is set to your control panel's dialog box, and the default volume is set to the System Folder of the current startup disk. Your control device function must preserve all of these settings.

Although the Finder intercedes with the system software and performs services on behalf of your control device function, it is your control device function's responsibility to detect and, if possible, recover from any error conditions. To avoid a memory error condition, your function should ensure that enough memory is available to handle the message from the Finder. On entry, the `main` function calls its `MyRoomToRun` procedure to perform this check.

The next sections describe how to handle each message passed in the `message` parameter.



## Determining If a Control Panel Can Run on the Current System

If you want your control device function to determine if your control panel can run on the current system, specify the values in your machine resource accordingly (see Table 8-1 on page 8-21). In this case, the Finder calls your function for the first time with a `macDev` message. The Finder calls your control device function with a `macDev` message only once.

In response to the `macDev` message, your control device function can check the hardware configuration of the current system. As necessary, your control device function should determine which computer it is being run on, what hardware is connected, and what is installed in the slots, if there are slots. The application-defined `MyCheckMachineCharacteristics` procedure, used in Listing 8-8 on page 8-27, performs these checks for the River control panel. Your control device function should return either a 0 or a 1 as its function result in response to the `macDev` message. These values have specific meanings in response to a `macDev` message, and the Finder does not interpret them as error codes. If your control panel file can run on the current system, return a function result of 1; if your control panel file cannot run on it, return a function result of 0. If your function returns a result of 0, the Finder does not open your control panel; instead, it displays an alert box to the user.

**Note**

If your machine resource specifies that your control panel runs on all systems, or if the machine resource identifes the restrictions that apply to your control panel, the Finder does not call your control device function with a `macDev` message. ◆

## Initializing the Control Panel Items and Allocating Storage

If your control panel can run on the current system, the Finder calls your control device function and specifies `initDev` in the `message` parameter. Except for a `macDev` message, your control device function should not process any other messages before it receives and successfully processes an `initDev` message. In response to an `initDev` message, your function should allocate any private storage it needs to implement its

features, initialize the settings of controls in the control panel, and perform any other necessary initialization tasks.

Because control panels cannot use normal global variables to retain information once the control device function returns, the interface between the Finder and the control device function provides a way to preserve memory that your control device function might allocate. If, for example, your control device function allocates memory to save data between calls, you return a handle to the allocated memory as the function result in response to the Finder's `initDev` message. The next time it calls your function, the Finder passes this handle back as the value of the `cdevStorageValue` parameter. After sending an `initDev` message, the Finder always passes to your function the function result previously returned as the value of the `cdevStorageValue` parameter. In this way, the Finder makes the handle available to your function, until your function returns an error code.

When the Finder calls your function with the `initDev` message, it passes the constant `cdevUnset` in the `cdevStorageValue` parameter; this value indicates that your function has not allocated any memory. If you do not create a handle and allocate memory in response to the `initDev` message, you should return this value (`cdevUnset`) as your function result. In this case, the Finder continues to pass this value to your control device function, and your function should continue to return this value until your control device function encounters an error.

Before the Finder calls your function with an `initDev` message, it has already drawn the dialog box and any items defined in your item list resource, except for user items. During initialization, you set the default value for any controls, and, if necessary, draw any user items. You can store the default values for controls in a resource located in a preferences file within the Preferences folder. To initially set the values for your panel's controls (such as radio buttons and checkboxes) and editable text, retrieve the default values from the resource and then use the Dialog Manager's `GetDialogItem` procedure and the Control Manager's `SetControlValue` procedure.

The Finder calls QuickDraw to draw the static text for your control panel. QuickDraw uses the default application font for this purpose; for Roman scripts, this is 9-point Geneva. For System 7, you can include a font information (`'finf'`) resource in your control panel file to specify a font to be used for static text.

For example, you can use a font information resource to specify 12-point Chicago, which is the recommended font for Roman scripts. You can also use an `'finf'` resource to change the font of static text in control panels localized for other system scripts. If you include an `'finf'` resource, the Finder sets the font, font style, and font size for the graphics port to the values you specify and uses these values to draw any static text. See "Specifying the Font of Text in a Control Panel" on page 8-23 for more information.

**Note**
The Control Manager uses the system font for text strings that are part of a control item. ◆

Listing 8-9 shows the `MyInitializeCP` procedure, which the River control device function calls to handle the `initDev` message. This procedure calls the `NewHandle` function to create a handle to a record of type `MyRiverStorage` (see Listing 8-8 on page 8-27). The procedure then initializes the fields of this record. It also calls its own procedure, `MyGetUserPreferenceSettings`, which reads a resource file containing the initial settings for the controls. This resource contains either the original default values or new values set by the user from the control panel.

The `MyInitializeCP` procedure sets initial values for any controls in its control panel. For each control, `MyInitializeCP` calls the Dialog Manager's `GetDialogItem` procedure to get a handle to the control and then calls the Control Manager's `SetControlValue` procedure to restore the last setting of the control. The first time a user uses the control panel, the initial values are the default values; after that, the initial values are those last set by the user. The `MyInitializeCP` procedure restores the last settings of radio buttons and checkboxes, sets the menu item to the last item chosen by the user in pop-up menus, and restores the text that the user last entered in editable text items.

Finally, the `MyInitializeCP` procedure returns in the `cdevStorageValue` parameter a handle to the memory it has allocated. The control device function then returns this value as its function result. In all subsequent calls to the control device function, the Finder passes this value back in the `cdevStorageValue` parameter.

The River control panel uses the memory it allocates to save values indicating that the user has changed a setting. When the user clicks the Show Me button or closes the control panel, the control device function notifies the River screen saver system extension that the settings have changed. The River screen saver then uses the new settings when it displays the river on the screen.

**Listing 8-9** Initializing a control panel: Allocating memory and setting controls

```
PROCEDURE MyInitializeCP (VAR cdevStorageValue: LongInt; CPDialog: DialogPtr;
                          VAR myRiverHndl: MyRiverStorageHndl);
VAR
   initOnSetting, initOffSetting, initUphillSetting, initDownhillSetting,
   initCircularSetting, initBabbleSetting, initRiverColorSetting: Integer;
   initVelocityText:    Str255;
   startSel, endSel:    Integer;
   itemType:            Integer;
   itemHandle:          Handle;
   itemRect:            Rect;
```

```
BEGIN
   myRiverHndl := MyRiverStorageHndl(NewHandle(Sizeof(MyRiverStorage)));
   IF myRiverHndl <> NIL THEN
   BEGIN    {initialize fields in myRiver record}
      myRiverHndl^^.count := 0;
      myRiverHndl^^.err := 0;
      myRiverHndl^^.settingsChanged := FALSE;
   END;
   {set default or saved values for each setting in this control panel-- }
   { usually a control panel reads these values from a resource file}
   MyGetUserPreferenceSettings(initOnSetting, initOffSetting,
                               initUphillSetting, initDownhillSetting,
                               initCircularSetting, initBabbleSetting,
                               initRiverColorSetting, initVelocityText,
                               startSel, endSel);
   {set the initial values of buttons and other controls using the Dialog }
   { Manager's GetDialogItem & the Control Mgr's SetControlValue procedures}

   GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle, itemRect);
   SetControlValue(ControlHandle(itemHandle), initOnSetting);

   GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle, itemRect);
   SetControlValue(ControlHandle(itemHandle), initOffSetting);

   GetDialogItem(CPDialog, kUphillRadButton, itemType, itemHandle, itemRect);
   SetControlValue(ControlHandle(itemHandle), initUphillSetting);

   GetDialogItem(CPDialog, kDownhillRadButton, itemType,itemHandle,itemRect);
   SetControlValue(ControlHandle(itemHandle), initDownhillSetting);

   GetDialogItem(CPDialog, kCircularRadButton, itemType,itemHandle,itemRect);
   SetControlValue(ControlHandle(itemHandle), initCircularSetting);

   GetDialogItem(CPDialog, kBabbleCheckBox, itemType, itemHandle, itemRect);
   SetControlValue(ControlHandle(itemHandle), initBabbleSetting);

   GetDialogItem(CPDialog, kRiverColorMenu, itemType, itemHandle, itemRect);
   SetControlValue(ControlHandle(itemHandle), initRiverColorSetting);

   GetDialogItem(CPDialog, kVelocityEditText, itemType, itemHandle,itemRect);
   SetDialogItemText(itemHandle, initVelocityText);
   SelectDialogItemText(CPDialog, kVelocityEditText, startSel, endSel);
   cdevStorageValue := Ord4(myRiverHndl);
 END;
```

If you define your text items as user items, your control device function must draw the text in response to an `initDev` message. See "Handling Text Defined as User Items" on page 8-43 for details.

## Responding to Activate Events

When a control panel is active, your control device function is responsible for making each control active or inactive, as appropriate. For example, your function should draw a bold outline around the control panel's default button. By contrast, when your control panel is inactive, your control device function should make all its controls inactive, causing the Control Manager to draw them in gray or in grayscale, depending on the bit depth of the monitor. This provides a visual indication to the user that a control panel is inactive, and it distinguishes the active window from inactive ones.

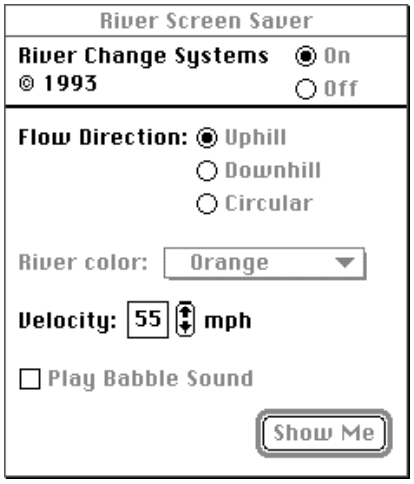
Whenever the Event Manager generates an activate event for your control panel in response to a user action, the Finder intercepts the activate event and calls your control device function with either an `activDev` message or a `deActivDev` message. In either case, the Finder passes to your function, in the parameter `theEvent`, the event record for the activate event and, in the `cdevStorageValue` parameter, a handle to the memory previously allocated by your function.

For example, the Finder calls your control device function with an `activDev` message (after sending an `initDev` message) when the user opens your control panel or clicks your inactive control panel after using another control panel or an application. Your function should respond to an `activDev` message by drawing a bold outline around the default button. It should also make the default button and any other controls in your control panel active. You can use the Control Manager's `HiliteControl` procedure to make a control active or inactive. (See the chapter "Control Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about `HiliteControl`.)

In general, your function does not need to update user items in response to an activate event, apart from drawing a bold outline around the default button. If, however, your control panel includes a user item that requires updating, such as a clock that shows the current time, your control device function should update that user item.

The Finder calls your control device function with a `deActivDev` message when the user clicks another control panel, runs an application, or otherwise brings another window to the front. In this case, your function should respond by drawing the outline of the default button in gray and making inactive any other controls in your control panel. While a control is inactive, the Control Manager does not respond to mouse events in it. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to make buttons, radio buttons, checkboxes, and pop-up menus inactive and active in response to activate events. Figure 8-10 shows the River control panel when it is inactive. Note that all of its controls are dimmed.

**Figure 8-10** Example of an inactive control panel




The River control device function calls its own procedure to handle both `activDev` and `deActivDev` messages. Listing 8-10 shows the `MyActivateControlPanel` procedure, which either makes the controls active in response to an `activDev` message or inactive in response to a `deActivDev` message.

In response to activate events, this procedure calls the Dialog Manager's `GetDialogItem` procedure to get a handle to the default button and then calls the Control Manager's `HiliteControl` procedure to make the control active. To draw the bold outline around the default button, the `MyActivateControlPanel` procedure calls its own procedure, `MyDrawDefaultButtonOutline`. (See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for detailed instructions on drawing an outline around a default button.) The procedure then makes all other controls active.

In response to a `deActivDev` message, the `MyActivateControlPanel` procedure makes all its controls inactive. In addition, it uses its own procedure, `MyDrawDefaultButtonOutline`, to draw a gray outline around the default button.

**Note**
If the dialog box uses a color graphics port, you can use the Color QuickDraw function `GetGray` to return a blended gray based on the foreground and background colors. ◆

**Listing 8-10** Responding to an activate event

```
PROCEDURE MyActivateControlPanel (VAR cdevStorageValue: LongInt;
                                  CPDialog: DialogPtr;
                                  myRiverHndl: MyRiverStorageHndl;
                                  activate: Boolean);
VAR
   itemType:      Integer;
   itemHandle:    Handle;
   itemRect:      Rect;
BEGIN
   IF activate THEN
   BEGIN    {control panel becoming active}
      {activate the default button (ShowMe) and draw bold outline around it}
      GetDialogItem(CPDialog, kShowMe, itemType, itemHandle, itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);
      MyDrawDefaultButtonOutline(CPDialog, kShowMe);

      {make other controls active}
      GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle, itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);

      GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle, itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);

      GetDialogItem(CPDialog, kUphillRadButton,itemType,itemHandle,itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);

      GetDialogItem(CPDialog, kDownhillRadButton, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);

      GetDialogItem(CPDialog, kCircularRadButton, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);
```

```
      GetDialogItem(CPDialog, kBabbleCheckBox, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);

      GetDialogItem(CPDialog, kRiverColorMenu, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 0);
   END
   ELSE
   BEGIN    {control panel becoming inactive}
      {make the default button inactive and draw gray outline around it}
      GetDialogItem(CPDialog, kShowMe, itemType, itemHandle, itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);
      MyDrawDefaultButtonOutline(CPDialog, kShowMe);

      {make other controls inactive}
      GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle, itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);

      GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle, itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);

      GetDialogItem(CPDialog,kUphillRadButton,itemType,itemHandle,itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);

      GetDialogItem(CPDialog, kDownhillRadButton, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);

      GetDialogItem(CPDialog, kCircularRadButton, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);

      GetDialogItem(CPDialog,kBabbleCheckBox,itemType,itemHandle,itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);

      GetDialogItem(CPDialog,kRiverColorMenu,itemType,itemHandle,itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);
   END;
END;
```

#### Using Multiple Dialog Boxes

The use of nested dialog boxes is not recommended in control panels. If you decide to use them nevertheless, keep in mind that the Finder may send your control device function a `deActivDev` message before your code that displays and initializes the second dialog box completes. This is because when your control device function calls `DialogSelect` to handle an event in a second dialog box, `DialogSelect` issues a call to `GetNextEvent`. In turn, the system software sends to the Finder an activate event instructing it to deactivate the main control panel's dialog box. However, this situation should not cause unusual problems, and your code should handle the `deActivDev` message, then continue its processing for the second dialog box. ◆

## Responding to Keyboard Events

The Finder intercepts all key-down and auto-key events for your control panel. The Finder sends your control device function a keyboard event through the `keyEvtDev` message for all keystrokes except Command-key equivalents. The Finder processes all Command-key equivalents on behalf of your control panel except those that it maps to its own Edit menu commands. The Finder converts these Command-key equivalents to messages and passes them on (as `cutDev`, `copyDev`, `pasteDev`, `undoDev`, and `clearDev` messages) to your control panel for processing. (See "Handling Edit Menu Commands" on page 8-46 for more information.)

#### Note

In System 6, the Control Panel desk accessory does not convert Command-key equivalents for Edit menu commands to edit messages; instead it passes the Command-key equivalent to your control device function as a `keyEvtDev` message. For backward compatibility, when your control device function receives a `keyEvtDev` message, it should check for Command-key equivalents as follows: it should examine the `modifiers` field and the `message` field of the event record to identify the Command-key equivalent, process it, and set the event record's `what` field to `nullEvent`. In this way, you prevent the Control Panel desk accessory from passing the keystroke to TextEdit for further handling. Listing 8-11 illustrates this technique. ◆

In addition to handling Command-key equivalents, your control device function should respond appropriately when the user presses the Enter key or the Return key. In either case, your function should map the keypress to your control panel's default button, if any, and perform the action corresponding to that button. For instance, the `MyHandleKeyEvent` procedure shown in Listing 8-11 calls its `MyShowMe` routine whenever the user presses Enter or Return. This routine signals the River system extension to display the river on the screen.

Your control device function does not need to process most other keystrokes. The Finder passes keyboard events on to `DialogSelect`, which calls TextEdit to handle text entry in editable text items. However, in some cases you might want your function to process the keypress and return the constant `nullEvent` in the `what` field of the event record. For example, if your control panel includes an editable text item that accepts only numeric characters, your function can detect an invalid value, signal the user by beeping, then modify the `what` field to prevent the Finder from passing the event to the Dialog Manager. Listing 8-11 illustrates this technique: the user can enter only numeric values in the Velocity editable text item.

**Listing 8-11** Responding to a keyboard event

```
PROCEDURE MyHandleKeyEvent (VAR theEvent: EventRecord; CPDialog: DialogPtr;
                            message: Integer);
VAR
   theChar:    Char;
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
   finalTicks: LongInt;
BEGIN
   {in System 6, you need to check for Command-key equivalents}
   {get the character from the message field of the event record}
   theChar := CHR(BAnd(theEvent.message, charCodeMask));
   IF BAnd(theEvent.modifiers, cmdKey) <> 0 THEN
   BEGIN {Command key down}
      theEvent.what := nullEvent; {change the event to a null event so that }
                                  { TextEdit will ignore it}
      CASE theChar OF
         'X', 'x':
            message := cutDev;
         'C', 'c':
            message := copyDev;
         'V', 'v':
            message := pasteDev;
         OTHERWISE
            message := nulDev; {ignore any other Command-key equivalents}
      END;  {of CASE}
      MyHandleEditCommand(message, CPDialog);
   END; {of command-key down}
   CASE theChar Of
      '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
      ;  {valid input, let DialogSelect/TextEdit handle key input}
```

```
      OTHERWISE
      BEGIN
         IF (theChar = Char(kCRkey)) OR (theChar = Char(kEnterKey)) THEN
         BEGIN {user pressed Return or Enter, map to default button}
            GetDialogItem(CPDialog, kShowMe, itemType, itemHandle, itemRect);

            HiliteControl(ControlHandle(itemHandle), inButton);
            Delay(8, finalTicks);
            HiliteControl(ControlHandle(itemHandle), 0);
            MyShowMe(CPDialog); {perform action defined by default button}
            theEvent.what := nullEvent;
         END {of Return or Enter}
         ELSE IF (theChar = Char(kDeleteKey)) THEN
            {let DialogSelect/TextEdit handle it}
         ELSE
         BEGIN    {invalid input, don't allow this character as input}
            SysBeep(40);
            theEvent.what := nullEvent;
         END;
      END; {of otherwise}
   END;  {of CASE}
END;
```

## Responding to Mouse Events

When the user clicks any active, enabled controls in your control panel, system software generates a mouse event. The Finder intercepts this event and passes it to your control device function as a `hitDev` message.Your control device function typically changes the setting of the control or performs the appropriate action in response to a `hitDev` message.

Along with the `hitDev` message, the Finder passes three values that your control device function uses to determine which item the user clicked.

- In the `CPDialog` parameter, the Finder passes a pointer to your control panel's dialog box.
- In the `item` parameter, the Finder passes the number of the item, as defined in your item list, that the user clicked.
- In the `numItems` parameter, a value provided for backward compatibility with the Control Panel desk accessory, the value passed depends on the system currently in effect. In System 6, this number is the number of items in the item list of the Control Panel desk accessory. In System 7, the Finder always passes a value of 0 in `numItems`.

In System 6, the Control Panel desk accessory uses the `numItems` parameter to pass the number of items in its own item list. The Control Panel desk accessory appends your control panel's item list to its own. To get the correct number of the clicked item, you need to subtract the number of items in the desk accessory's item list (`numItems`) from

the number passed in the `item` parameter. Although the `numItems` parameter contains 0 in System 7, to maintain backward compatibility, you should always determine an item number by subtracting the value of `numItems` from the value of `item`. If you do so, your control panel can operate correctly with both the Finder and the Control Panel desk accessory. For more information about item lists, see "Creating the Item List Resource" on page 8-17, and the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

The River control device function determines the correct item number in its `CASE` statement before it calls its `MyHandleHitInDialogItem` procedure to handle the `hitDev` message. Here is the code segment, also shown in Listing 8-8 on page 8-27, that determines the item number:

```
hitDev: {user clicked dialog item}
 BEGIN
   item := item - numItems;
   MyHandleHitInDialogItem(item, cdevStorageValue,
                           CPDialog, myRiverHndl);
END;
```

Listing 8-12 shows the River control panel's `MyHandleHitInDialogItem` procedure, which takes the appropriate action in response to the item the user clicked. For the Show Me button, the procedure calls its `MyShowMe` procedure, which instructs its system extension to display the River screen saver using any new values.

For the On and Off radio buttons, `MyHandleHitInDialogItem` first calls the Dialog Manager's `GetDialogItem` procedure to get a handle to each radio button and then the Control Manager's `GetControlValue` function to determine the current setting. If the radio button clicked was previously off, `MyHandleHitInDialogItem` reverses its setting and also reverses the setting of the radio button that was previously on. If the user clicks any one of the group of radio buttons governing flow direction (Uphill, Downhill, Circular), `MyHandleHitInDialogItem` calls another application-defined routine, the `MyHandleFlowRadioButton` procedure. Although not shown in this listing, this procedure handles each of the three radio buttons, checking whether a button's value has changed and, if so, resetting the control.

If the user clicked the Play Babble Sound checkbox, `MyHandleHitInDialogItem` reverses its setting.

The River control panel defines two user items that enclose the up arrow and the down arrow. If the user clicks either of these areas, `MyHandleHitInDialogItem` calls its own `MyHandleHitInArrows` procedure to handle this event. The routine either increments or decrements the number displayed in its editable text item accordingly.

The River control panel ignores clicks in any other item, because the Dialog Manager automatically handles clicks in pop-up controls and editable text items.

After handling the `hitDev` message, `MyHandleHitInDialogItem` sets the `settingsChanged` field of the `MyRiverStorage` record. Other routines use this value

to determine if the preferences file needs updating or if its system extension needs to read the preferences file and use the new values when displaying the screen saver.

**Listing 8-12** Responding to the user's interaction with controls

```
PROCEDURE MyHandleHitInDialogItem (item: Integer;
                                   VAR cdevStorageValue: LongInt;
                                   CPDialog: DialogPtr;
                                   myRiverHndl: MyRiverStorageHndl);
VAR
   newOnSetting, newOffSetting:  Integer;
   newUphillSetting, newDownhillSetting, newCircularSetting: Integer;
   newBabbleSetting:             Integer;
   newVelocityText:              Str255;
   newRiverColorSetting:         Integer;
   itemType:                     Integer;
   itemHandle:                   Handle;
   itemRect:                     Rect;
BEGIN
   CASE item OF
    kShowMe:
      MyShowMe(CPDialog);
    kOnRadButton:
      BEGIN
         {get handle to the On radio button, get its current value, }
         { and then if it was off, change it to on}
         GetDialogItem(CPDialog,kOnRadButton,itemType,itemHandle,itemRect);
         newOnSetting := GetControlValue(ControlHandle(itemHandle));
         IF (newOnSetting = 0) THEN
         BEGIN
            newOnSetting := 1 - newOnSetting;
            SetControlValue(ControlHandle(itemHandle), newOnSetting);
            {get handle to the Off radio button, get its current value, }
            { and then change it}
            GetDialogItem(CPDialog, kOffRadButton, itemType, itemHandle,
                          itemRect);
            newOffSetting := 1 - newOnSetting;
            SetControlValue(ControlHandle(itemHandle), newOffSetting);
         END;
      END;
    kOffRadButton:
      BEGIN
         {get handle to the Off radio button, get its current value, }
```

```
          { and then if it was off, change it to on}
         GetDialogItem(CPDialog,kOffRadButton,itemType,itemHandle,itemRect);
         newOffSetting := GetCtlValue(ControlHandle(itemHandle));
         IF (newOffSetting = 0) THEN
         BEGIN
            newOffSetting := 1 - newOffSetting;
            SetControlValue(ControlHandle(itemHandle), newOffSetting);
            newOffSetting := GetCtlValue(ControlHandle(itemHandle));
            {get handle to the On radio button, get its current value, }
            { and then change it}
            GetDialogItem(CPDialog, kOnRadButton, itemType, itemHandle,
                          itemRect);
            newOnSetting := 1 - newOffSetting;
            SetControlValue(ControlHandle(itemHandle), newOnSetting);
         END;
       END;
     kUpHillRadButton, kDownHillRadButton, kCircularRadButton:
       {this routine handles the Flow Direction radio buttons}
       MyHandleFlowRadioButton(item, CPDialog);
     kBabbleCheckBox:
       BEGIN
          {get handle to Play Babble Sound checkbox, get its current value, }
          { and then change it}
          GetDialogItem(CPDialog, kBabbleCheckBox, itemType, itemHandle,
                        itemRect);
          newBabbleSetting := GetControlValue(ControlHandle(itemHandle));
          newBabbleSetting := 1 - newBabbleSetting;
          SetControlValue(ControlHandle(itemHandle), newBabbleSetting);
       END;
     kUserItemUpArrow, kUserItemDownArrow:
       MyHandleHitInArrows(item, CPDialog);
   END; {of CASE}
   myRiverHndl^^.settingsChanged := TRUE;
END;
```

## Responding to Update Events

Whenever the Event Manager generates an update event for your control panel, the Finder intercepts the update event and calls your control device function with an `updateDev` message. Your control device function should perform any updating necessary, apart from the standard dialog item updating that the Dialog Manager performs. An update event gives your control device function the opportunity to redraw user items that might require updating, such as a clock. You should also redraw the outline around your default button in response to an update event. Notice that the `MyUpdateControlPanel` procedure in Listing 8-13 does this by calling its `MyDrawDefaultButtonOutline` procedure, which the control device function also calls in response to an `activDev` or `deActivDev` message. If your control panel has an editable text item, you don't need to include code to make the caret blink. The Dialog Manager calls `TEIdle` for this purpose.



**Listing 8-13** Responding to update events

```
PROCEDURE MyUpdateControlPanel (VAR cdevStorageValue: LongInt;
                                CPDialog: DialogPtr;
                                myRiverHndl: MyRiverStorageHndl);
BEGIN
   {draw the outline around the default button on an update event}
   MyDrawDefaultButtonOutline(CPDialog, kShowMe);
END;
```

## Handling Text Defined as User Items

If you want to use a font other than the default application font for your control panel's text, you should either include an `'finf'` resource in your control panel file and define your text as static text items or define your text using user items. See "Creating a Font Information Resource" on page 8-23 for details on changing the font using an `'finf'` resource. This section gives details on how to define text using user items. You might want to use this approach so that your control panel can run in the Finder and the Control Panel desk accessory.

If you define the text in your control panel using user items, you need to draw the text in response to an `updateDev` message, just as you would any other user item that requires updating. (You draw the text initially in response to an `initDev` message.)

For each item, this process entails

- Setting the text font, style, and size fields to be used. (You use the QuickDraw procedures `TextFont`, `TextFace`, and `TextSize` for this purpose.)
- Positioning the pen where you want to draw the text. You draw the text in the rectangle defined for it in the item list resource. (You can use the QuickDraw procedure `MoveTo` to set the initial location of the pen.)
- Drawing the text string. (You can use the QuickDraw `DrawString` procedure for this purpose.)

Listing 8-14 shows the `MyDrawText` procedure. The River control device function might use this procedure to draw any text that it defined as user items. First the `MyDrawText` procedure calls the QuickDraw `TextFont`, `TextFace`, and `TextSize` procedures to set the graphics port font to 12-point Chicago.

Then, for each text item, `MyDrawText` calls it own `MyGetUserText` procedure to get the text string and the coordinates of the text string as defined by the display rectangle of the user item. (See"Defining Text in a Control Panel as User Items" on page 8-24 for details about the item list.) Next, `MyDrawText` calls the QuickDraw `MoveTo` procedure to position the pen and QuickDraw's `DrawString` procedure to draw the text.

**Listing 8-14** Drawing text defined as user items

```
PROCEDURE MyDrawText;
VAR
   textForUserItem:  Str255;
   textH, textV:     Integer;
BEGIN
   TextFont(0);      {set the font to the system font (Chicago)}
   TextFace([]);     {set the text face to normal}
   TextSize(12);     {set the font size to 12-point}
   {get the text and location for the first text string}
   MyGetUserText(kFlow, textForUserItem, textH, textV);
   MoveTo(textH, textV);
   DrawString(textForUserItem);     {draw the text}
   {get the text and location for the next text string}
   MyGetUserText(kVelocity, textForUserItem, textH, textV);
   MoveTo(textH, textV);
   DrawString(textForUserItem);     {draw the text}
   {get the text and location for the next text string}
   MoveTo(textH, textV);
   MyGetUserText(kMph, textForUserItem, textH, textV);
   DrawString(textForUserItem);     {draw the text}
END;
```

## Responding to Null Events

Whenever the Event Manager generates a null event for your control panel, the Finder intercepts the event and calls your control device function with a `nulDev` message. Your control device function should respond to a `nulDev` message by performing any needed idle processing. However, your control device function should do minimal processing in response to a null event; for example, it should not refresh control settings.

## Responding to the User Closing the Control Panel

When the user closes your control panel, the Finder calls your control device function with a `closeDev` message, signaling it to terminate gracefully. In response to this message, your control device function must dispose of any memory it has allocated, including any pointers or handles it has allocated.

Before your function begins this process, however, it can perform other needed tasks. For example, the River control device function checks whether the user changed the values of any settings. If so, it updates its preferences file to reflect the changes.

Listing 8-15 shows the `MyCloseControlPanel` procedure, which the River control device function calls to handle the `closeDev` message. The `MyCloseControlPanel` procedure checks the `settingsChanged` field of its `MyRiverStorage` record to determine if the user changed the settings (the control device function sets this field whenever the user changes a setting). If necessary, `MyCloseControlPanel` calls a procedure to update the preferences file with the new values stored in the `MyRiverStorage` record. Next, `MyCloseControlPanel` disposes of the memory that the control device function previously allocated by disposing of the handle in the `myRiverHndl` parameter. It then sets the `cdevStorageValue` parameter to 0. The control device function returns this value as its function result.

**Listing 8-15** Terminating a control device function when the user closes the control panel

```
PROCEDURE MyCloseControlPanel (myRiverHndl: MyRiverStorageHndl;
                               VAR cdevStorageValue: LongInt);
BEGIN
   {if the user changed any of the settings, }
   { write the new settings to the River preferences file}
   IF myRiverHndl^^.settingsChanged THEN
      MyWriteUserPreferences(myRiverHndl);
   {dispose of any allocated storage}
   IF myRiverHndl <> NIL THEN
   BEGIN
      DisposeHandle(Handle(myRiverHndl));
      cdevStorageValue := 0;
   END;
END;
```

## Handling Edit Menu Commands

Although you cannot implement a menu bar in your control panel, the user can choose the Finder's Edit menu Undo, Cut, Copy, Paste, and Clear commands when working in an editable text item. When the user chooses one of these commands from the Edit menu or presses its Command-key equivalent, the Finder maps the command to a message and calls your control device function with the message. The values in the `message` parameter for these commands are `undoDev` for Undo, `cutDev` for Cut, `copyDev` for Copy, `pasteDev` for Paste, and `clearDev` for Clear.

**Note**

In System 6, the Control Panel desk accessory does not convert Command-key equivalents for Edit menu commands to edit messages; instead it passes the Command-key equivalent to your control device function as a `keyEvtDev` message. See "Responding to Keyboard Events" beginning on page 8-37 for details on handling keyboard events, including Command-key equivalents. ◆

Listing 8-16 show the `MyHandleEditCommand` procedure. The River control device function calls this procedure from within its `CASE` statement to handle an edit message. For the Cut, Copy, and Clear commands, `MyHandleEditCommand` calls Dialog Manager routines to perform the desired operation. For the Paste command, `MyHandleEditCommand` first uses its `MyCheckLength` function to ensure that the length of any text to be pasted does not exceed the TextEdit text buffer limit of 32 KB; only then does it call `DialogPaste`. The Dialog Manager calls TextEdit to perform the operation.

**Listing 8-16** Responding to Edit menu commands

```
PROCEDURE MyHandleEditCommand (message: Integer;
                               CPDialog: DialogPtr);
BEGIN
   CASE message OF
      cutDev:              {use Dialog Manager to cut the text}
         DialogCut(CPDialog);
      copyDev:             {use Dialog Manager to copy the text}
         DialogCopy(CPDialog);
      clearDev:            {use Dialog Manager to clear the text}
         DialogDelete(CPDialog);
      pasteDev:
       BEGIN               {check length, then paste the text}
         IF MyCheckLength(CPDialog) THEN
            DialogPaste(CPDialog);
       END;
   END;   {of CASE}
END;
```

## Handling Errors

Your control device function is responsible for detecting and, if possible, recovering from error conditions. If your function cannot recover from an error condition, it must dispose of any memory that it previously allocated, restore the system stack, and return as its function result one of three error codes.

If your control panel encounters an error due to missing resources or lack of memory, your control device function should return `cdevResErr` or `cdevMemErr`. When the Finder receives either of these error codes, it closes the control panel and displays an alert box reporting the problem.

Your control device function should return a generic error code (`cdevGenErr`) for all other errors. When the Finder receives this generic error code, it closes the control panel but does not display an alert box; if it can do so, your function should display an alert box to the user before completing. Your function can also return this error code to signal a missing-resources or lack-of-memory error. Use this error code instead of `cdevResErr` or `cdevMemErr` if you want your function, not the Finder, to display a meaningful error message that directs the user in resolving the problem.

The Finder in System 7 and the Control Panel desk accessory in System 6 respond differently to any error codes that your control panel returns. In System 6, after your control device function terminates, the Control Panel desk accessory fills the area previously occupied by your control panel with the background pattern, in effect dimming it. The Control Panel desk accessory dialog box remains open because the user can use other control panels represented in the icon list. Your control panel's area remains dimmed until the user selects another control panel.

Table 8-2 shows the constants defined for these error codes and the corresponding responses by the Finder and the Control Panel desk accessory.

**Table 8-2** Error codes and their meaning

| Constant | Value | Meaning |
|---|---|---|
| `cdevGenErr` | –1 | Generic error<br><br>In System 7, the Finder closes your control panel but does not display an alert box to the user.<br><br>In System 6, the Control Panel desk accessory dims your control panel's area in the Control Panel window and passes 0 in the `cdevStorageValue` parameter the next time it calls your function. |
| `cdevMemErr` | 0 | Insufficient memory<br><br>In System 7, the Finder closes the control panel and displays an out-of-memory alert box to the user.<br><br>In System 6, the Control Panel desk accessory dims your control panel's area in the Control Panel window, displays an out-of-memory alert box to the user, and passes 0 in the `cdevStorageValue` parameter the next time it calls your function. |

*continued*

**Table 8-2** Error codes and their meaning (continued)

| Constant | Value | Meaning |
|---|---|---|
| `cdevResErr` | 1 | Missing resource<br><br>In System 7, the Finder closes the control panel and displays a missing-resources alert box to the user.<br><br>In System 6, the Control Panel desk accessory dims your control panel's area in the Control Panel window, displays a missing-resources alert box to the user, and passes 0 in the `cdevStorageValue` parameter the next time it calls your function. |

# Creating an Extension for the Monitors Control Panel

This section describes how to create an extension for the Monitors control panel. A monitors extension typically adds controls to the Options dialog box so that the user can set values for one or more features of a video card.

A monitors extension consists of a file of type `'cdev'` that contains the resources for a monitors extension, including a code resource of type `'mntr'`. This code resource, called a monitors extension function, communicates with the Monitors control panel, responding to requests from the Monitors control panel to handle events or perform actions. This section begins with a discussion of the interface components of a monitors extension. Then it describes how to

- create resources for your monitors extension, including how to define
  - a card resource to identify your monitors extension and display the name of your video card at the top of the Options dialog box
  - a rectangle resource to define the area in which to display your video card's controls
  - an item list resource to define additional items for display in the Options dialog box
- create and supply optional resources for your monitors extension
- write a monitors extension function

Before you develop an extension for the Monitors control panel, consider these three important points:

- You should develop a monitors extension *only* if you are the manufacturer of the video card for which you are providing the feature or features whose values the user can control.
- There can be only one extension to the Monitors control panel for each video card. Apple Computer, Inc., reserves the right to supply monitors extensions for its own video cards.
- If the features that you want to implement require an extensive or complex set of controls—for example, if you need to use nested dialog boxes—you should probably write a small application rather than an extension to the Monitors control panel.



## Designing the User Interface for a Monitors Extension

When the user clicks the Options button, the Monitors control panel displays the Options dialog box for the selected monitor. The Options dialog box contains standard controls that the Monitors control panel provides, such as the OK and Cancel buttons. Beneath these two buttons is a scrollable list of monitor types if the selected monitor belongs to a family of monitors. Beneath the icon is a scrollable list of gamma tables if the user is a **superuser** (a very knowledgeable user; a user indicates superuser status by pressing the Option key while clicking the Options button). These items are also defined by the Monitors control panel.

If you provide a monitors extension for your video card, the Monitors control panel adds any controls you define beneath the two scrollable lists, if one or both are displayed, or beneath the Cancel button. Figure 8-11 shows the Options dialog box for the Macintosh display card.

**Figure 8-11** An Options dialog box with standard controls

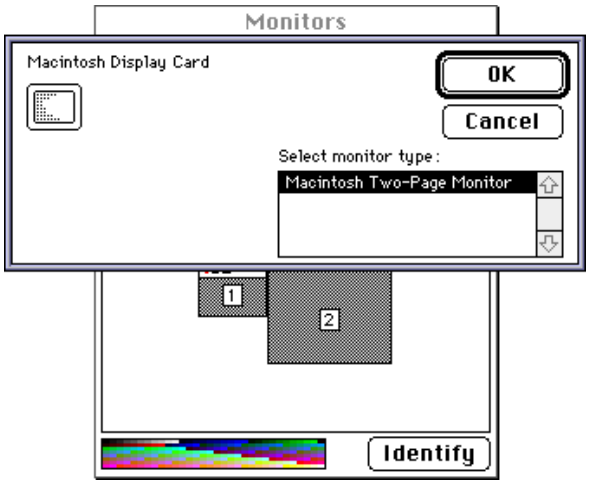
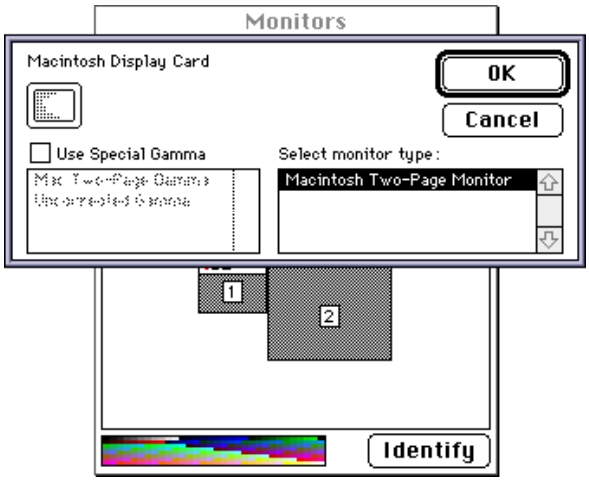


Figure 8-12 shows the Options dialog box for the Macintosh display card as it appears when the user presses the Option key while clicking the Options button.

**Figure 8-12** An Options dialog box with superuser controls




To provide the user interface for your video card's feature, you define a rectangle resource of type `'RECT'` specifying the amount of space you need to display your controls and an item list resource of type `'DITL'` specifying the controls themselves.

At the upper-left corner of the Options dialog box, the Monitors control panel displays the name of your video card and an icon representing it. The Monitors control panel defines the coordinates of these items. You must supply your video card's name in a required card resource of type `'card'` (see "Creating a Card Resource for a Monitors Extension" on page 8-51). You can optionally provide one or more members of an icon family (with resource ID –4064) that define an icon for your video card. If you do not provide icon resources with this resource ID for this purpose, the Monitors control panel displays the icon defined in the `sResource` data structure in the ROM on your video card. If your video card does not supply a default icon in the ROM, the Monitors control panel displays a generic monitors icon.

You can also supply an additional icon family to specify the icon that the Finder uses to represent your monitors extension file. The icon family resources are `'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`. When creating an icon for a monitors extension, design it so that it is square, except include at the bottom of the icon a tab-like form, indicating that the file the icon represents is an extension. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to create an icon family. Figure 8-13 shows an icon of type `'icl8'` for the monitors extension file supplied with the SurfBoard video card.

**Figure 8-13** The SurfBoard monitors extension icon




If you wish, you can design two sets of controls for your monitors extension: one set for ordinary users and one for superusers. When a user indicates superuser status by holding down the Option key while clicking the Options button in the Monitors control panel, the Monitors control panel notifies your monitors extension function to display the superuser controls. For more information, see "Creating an Item List Resource for a Monitors Extension" beginning on page 8-54.

## Creating the Required Resources for a Monitors Extension

This section describes the four required resources that you supply for your monitors extension.

To create these resources, either you can specify the resource description in an input file and compile the resource using a resource compiler, such as Rez, or you can directly create your resources in a resource file using a tool such as ResEdit.

The required resources and their resource IDs are

- the card (`'card'`) resource: resource ID from –4080 through –4065
- the rectangle (`'RECT'`) resource: resource ID –4096
- the item list (`'DITL'`) resource: resource ID –4096
- the monitor (`'mntr'`) resource: resource ID –4096

### Creating a Card Resource for a Monitors Extension

You create a card resource of type `'card'` to identify the monitors extension for your video card and to specify the name of your video card. When the monitor to which your card is connected is the selected one and the user clicks the Options button, the Monitors control panel checks all monitors extension files for a card resource that contains the name of your video card. If it finds a match, the Monitors control panel extends the Options dialog box to display the monitors extension containing the matching card resource. The Monitors control panel also displays, at the very top of the Options dialog box, your video card's name as defined in the card resource. This title indicates to the user that the Options dialog box pertains to your card. For example, Figure 8-11 on page 8-49 shows the Macintosh Display Card name at the top of the Options dialog box. The card resource is required, and its resource ID must be in the range of –4080 through –4065.

Your card resource must contain a Pascal string identical to the name of your video card as specified in the `sResource` data structure in the ROM of the card. (For more information on the `sResource` data structure, see *Designing Cards and Drivers for the Macintosh Family*, third edition.)

If you do not want to use the video card name specified in the ROM of the card, you can include in your monitors extension file a string list resource of type `'STR#'`. In that resource, specify an alternative name for the Monitors control panel to display. See "Providing an Alternative Name for a Video Card" on page 8-58 for more information.

You use a card resource to ensure that your monitors extension is called when the user selects the monitor to which your card is connected and clicks the Options button. Because your monitors extension file can contain as many card resources as you wish, one extension file can handle several types of video cards. For example, Listing 8-17 shows two card resources; thus, when the user selects the monitor connected to the SurfBoard Display Card or the SurfBoard Super Display Card, the monitors extension `MyMonExtend` is called. (See Listing 8-25 on page 8-64 for the `MyMonExtend` function.)

**Listing 8-17** Rez input for a card (`'card'`) resource

```
resource 'card' (-4080, purgeable)
{
   "SurfBoard Display Card"
};
resource 'card' (-4079, purgeable)
{
   "SurfBoard Super Display Card"
};
```

## Defining a Rectangle for a Monitors Extension

You create a rectangle resource of type `'RECT'` to define the display area for the controls of your monitors extension. When the user clicks the Options button in the Monitors control panel, the Monitors control panel uses your monitors extension to expand the Options dialog box under these circumstances: if the monitor connected to your video card is currently selected, and if you have provided a monitors extension for your card. Before displaying it, the Monitors control panel expands the Options dialog box to include the space defined by the rectangle resource. The rectangle resource is required, and its resource ID must be –4096.

To specify the top coordinate of your rectangle, determine the height in pixels of the space required to display your controls and specify that value as a negative number. For example, if you need a display area that is 60 pixels high, specify –60 as the top coordinate. Specify 0 as the left coordinate. This is the same value used to define the left edge of the Options dialog box, and your rectangle should have the same left edge.

Specify 0 as the bottom coordinate. You can think of the distance from the bottom coordinate to the top coordinate—60 pixels, in this example—as the height of your

rectangle. Specify 320 as the right coordinate. This is the same value used to define the right edge of the Options dialog box, and your rectangle should have the same right edge.

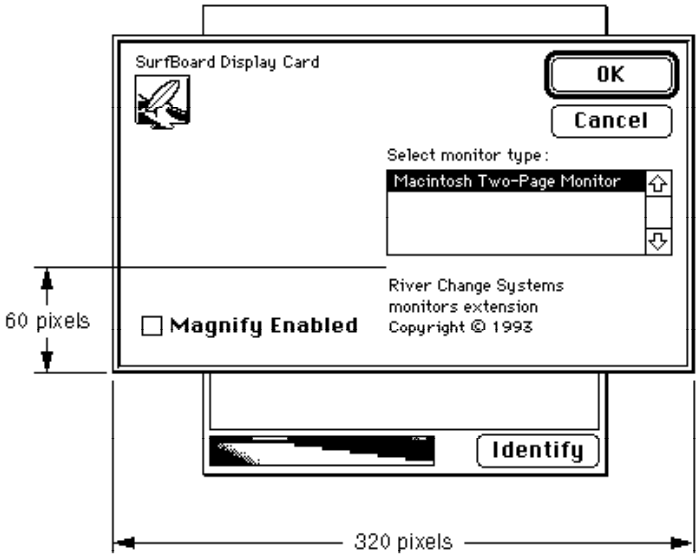
**Note**

Although you specify other coordinate values for your rectangle's origin, when you assign coordinates to your controls, assume that the origin of the local coordinate system for your dialog items is (0,0). ◆

Figure 8-14 shows the Options dialog box for the SurfBoard Display Card. The OK and Cancel buttons and the scrollable list for the monitor type are standard controls. The Magnify Enabled checkbox and three lines of text have been added by the SurfBoard monitors extension. This figure shows the height and width, in pixels, defined in the rectangle resource; this is the area required to display the additional controls.



**Figure 8-14** Display area defined by a rectangle resource




Listing 8-18 shows, in Rez input, the rectangle resource used in this example. Notice that the top coordinate is –60 and the bottom coordinate is 0. In other words, the space to be added to the Options dialog box is 60 pixels high.

**Listing 8-18** Rez input for a rectangle (`'RECT'`) resource

```
resource 'RECT' (-4096, purgeable)
{
   {-60,0,0,320}
};
```

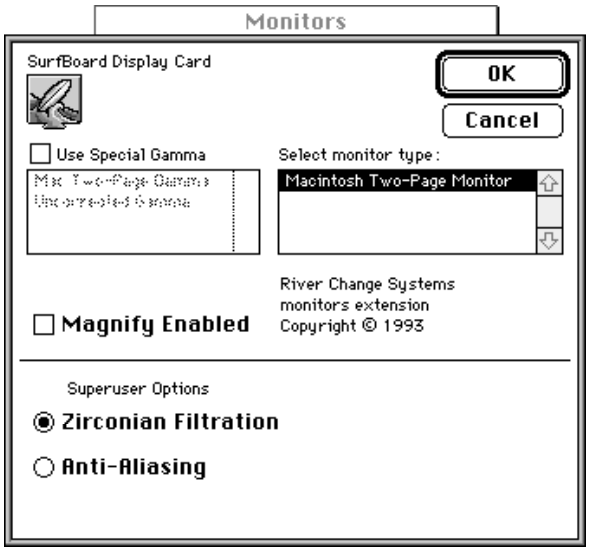
## Creating an Item List Resource for a Monitors Extension

You provide an item list resource of type `'DITL'` to specify which items you want to appear in the rectangle display area (see the previous section for information about the rectangle resource). In an item list, you specify static text, buttons, checkboxes, radio buttons, editable text, user items, the resource IDs of icons and QuickDraw pictures, and the resource IDs of other types of controls, such as pop-up menus. The item list is required, and its resource ID must be –4096.

When you assign coordinates to your controls, assume that the origin (that is, the upper-left corner) of the local coordinate system is (0,0). The Monitors control panel transforms the coordinates of your controls to the coordinate system that it uses for the Options dialog box. Thus, you must use the `GetDialogItem` procedure to get the true locations of your dialog items. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the `GetDialogItem` procedure.

If you add additional controls for superusers, you should place them below a horizontal line separating them from other controls, as illustrated in Figure 8-15.

**Figure 8-15** The SurfBoard Options dialog box with superuser controls



To draw a dividing line, specify a separate dialog item of type `userItem`. Listing 8-19 shows the item list resource for the SurfBoard monitor extension. Notice that the dividing line (item 2) is defined as a user item.

**Listing 8-19** Rez input for the SurfBoard monitors extension item list resource

```
resource 'DITL' (-4096, purgeable) {
   {
      /* [1] */
      {10, 151, 50, 314},
      StaticText {   disabled,   "River Change Systems\nmonitors extension"
                                 "\nCopyright © 1993"                    },
      /* [2] dividing line for superuser controls*/
      {60, 1, 61, 319},
      UserItem {     enabled                                             },
      /* [3] */
      {70, 28, 80, 236},
      StaticText {   enabled,    "Superuser Options"                     },
      /* [4] */
      {82, 7, 110, 200},
      RadioButton {  enabled,    "Zirconian Filtration"                  },
      /* [5] */
      {112, 7, 132, 200},
      RadioButton {  enabled,    "Anti-Aliasing"                         },
      /* [6] */
      {22, 7, 58, 160},
      CheckBox {     enabled,    "Magnify Enabled"                       }
      }
   };
```

Listing 8-29 on page 8-70 shows the procedure that the SurfBoard monitors extension function uses to draw a line separating the items for normal users from the items displayed for superusers. It uses the QuickDraw `FrameRect` procedure to draw the item as a 1-pixel-high rectangle. After calling the `FrameRect` procedure, a monitors extension can also dither the line in the same manner used to dither menu divider lines. (For information on the `FrameRect` procedure, see *Inside Macintosh: Imaging With QuickDraw*.)

If you use an item color table resource of type `'ictb'` to draw your items in color or in a different font, you must include placeholder entries for the standard Options dialog box items before you define the item color table entries to be mapped to the items in your monitors extension item list. This step is necessary because the Monitors control panel appends your monitors extension item list to that of the Options dialog box. To maintain the mapping between entries in the item color table (`'ictb'`) and your item list, you must account for the Options dialog box items.

Currently, the Options dialog box contains 10 items (although this number is subject to change in future implementations of the Monitors control panel). An item color table entry contains two words for each corresponding item. For this implementation of the Monitors control panel, you can ensure that the first item in your item list is mapped to the correct item color table entry as follows: create 10 entries in the item color table to correspond to the 10 items in the Options dialog box, and specify a value of 0 for both words of each entry.

### Creating the Monitor Code Resource

A monitor code resource (of type `'mntr'`) contains the code that carries out the functions of a monitors extension. In MPW, you can set the code resource type to `'mntr'` when you link the program. When you create such a resource, the resource must begin with a function that you provide, called the monitors extension function.

The Monitors control panel passes to your monitors extension function parameters that specify actions to perform. You can use the function result to keep a handle to allocated memory or to return an error code. For more information about the monitors extension function, see “Writing a Monitors Extension Function” beginning on page 8-61.

## Supplying Optional Resources for a Monitors Extension

Your monitors extension file can also include any of the optional resources described in this section. To create these resources, either you can specify the resource description in an input file and then use a resource compiler, such as Rez, to compile the resource, or you can use a tool such as ResEdit to create your resources in a resource file.

The optional resources and their resource IDs are

- The icon family resources (`'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`), which specify an icon for display in the upper-left corner of the Options dialog box: resource ID –4096.
- The version (`'vers'`) resources: resource ID 1 and 2.
- The string list (`'STR#'`) resource: resource ID –4096.
- The gamma table (`'gama'`) resource: resource ID from –4080 through –4065.

- The file reference (`'FREF'`) resource. The resource ID follows the normal conventions (typically, you assign a resource ID of 128).
- The bundle (`'BNDL'`) resource. The resource ID follows the normal conventions (typically, you assign a resource ID of 128).
- The icon family resources (`'ICN#'`, `'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, and `'ics4'`), which define the monitors extension file icon. The resource ID follows the normal conventions (typically, you assign a resource ID of 128).
- The system extension (`'INIT'`) resource.
- The signature resource: resource ID 0.

In addition to the optional resources that these sections describe, you can include private optional resources whose resource ID numbers must fall within the range –4080 through –4065.



## Specifying an Icon for the Options Dialog Box

To specify an icon that the Monitors control panel displays in the upper-left corner of the Options dialog box, you can define one or more members of an icon family. For each of these resources, you must assign a resource ID of –4064. If you provide an icon family, the Monitors control panel displays the appropriate icon according to the bit depth of the monitor. (Note that in System 6 you provide `'ICON'` or `'cicn'` icons instead of an icon family.) For more information on these icons, see the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.

If you do not supply either of these icon resources, the Monitors control panel displays the icon defined in the `sRsrcIcon` entry of the `sResource` data structure of your video card’s ROM. If your do not supply either of these resources and your video card does not include an icon, the Monitors control panel displays a generic icon that represents a monitor.

Listing 8-20 shows a partial listing of the icon family resources that define the SurfBoard video card icon shown in Figure 8-13 on page 8-51.

**Listing 8-20** Rez input for icon family resources for a monitors extension

```
data 'ICN#' (-4064, purgeable) {
   /*icon data goes here*/
};
data 'icl8' (-4064, purgeable) {
   /*icon data goes here*/
};
data 'icl4' (-4064, purgeable) {
   /*icon data goes here*/
};
```

## Specifying Version Information

You can include two kinds of version resources of type `'vers'` to provide version information for your monitors extension file. The version resource with a resource ID of 1 specifies the version of your monitors extension file. The version resource with a resource ID of 2 specifies the version of the group to which your file belongs—for example, the version number of the video card that your extension file supports.

The Finder displays version information about your monitors extension for the user. For complete information on how to specify the version resources and how the Finder displays the information from these resources in its information window, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Listing 8-21 shows a version resource with a resource ID of 1 that specifies the version number of the SurfBoard's monitor extension file. This version resource includes the copyright of the River Change Systems company, which manufactures the card.

**Listing 8-21** Rez input for a version (`'vers'`) resource

```
resource 'vers' (1) {
   0x01, 0x00, release, 0x00,
   0, /*verUS*/
   "1.00",
   "1.00, Copyright © 1993 River Change Systems."
};
```

## Providing an Alternative Name for a Video Card

The Monitors control panel displays the name of your video card in the upper-left corner of the Options dialog box. By default, it displays the name defined in the declaration ROM of the video card. To display a name for your video card that is different from the name in the declaration ROM of the video card, you can include a string list (`'STR#'`) resource with resource ID –4096. This resource must contain pairs of Pascal strings. The first string in each pair must be identical to the name of your video card as specified in the `sResource` data structure in the ROM of the card. (For more information on the `sResource` data structure, see *Designing Cards and Drivers for the Macintosh Family*, third edition.) The second string in each pair is the name that you want to display in the Options dialog box. You can have as many pairs of names in one string list resource as you wish; the Monitors control panel uses the first match it finds.

It is unlikely that you will need to override the name specified in the declaration ROM. However, if you have misspelled the card name on the board, or if you want to display a name that is more descriptive, you can include a string list resource. Listing 8-22 shows the string list resource for the SurfBoard video card monitors extension. This monitors extension includes two card resources (see Listing 8-17 on page 8-52), but the string list resource includes only one entry to override the name SurfBoard Super Display Card. In this example, when the Monitors control panel displays the Options dialog box for the SurfBoard Super Display Card, it displays the name SurfBoard Super Fast Display Card instead of the name in the card's declaration ROM.



**Listing 8-22** Rez input for the SurfBoard string list resource

```
resource 'STR#' (-4096, purgeable)
{
   {  "SurfBoard Super Display Card";
      "SurfBoard Super Fast Display Card"};
};
```

## Supplying Gamma Table Resources

To indicate status as a superuser, the user presses the Option key while clicking the Options button in the Monitors control panel. In response, the Monitors control panel displays a list of gamma tables (see Figure 8-12 on page 8-50).

The software driver for a video card uses a gamma table to correct for the fact that the intensity of each color on a video display is not linearly proportional to the intensity of the electron beam; in other words, the gamma table helps the video driver to provide the most accurate colors possible for a video display. Because the user might prefer a nonstandard color correction, many developers of video cards provide more than one gamma table for a given card.

To supply one or more gamma tables for a video card, include in the monitors extension file a named resource of type `'gama'` for each gamma table. To change the default gamma table for a monitor, the user clicks the Use Special Gamma checkbox and then selects a table by clicking its name in the list. The default gamma table for a monitor is the one listed in the screen resource of type `'scrn'`. For a complete discussion of gamma tables, see *Designing Cards and Drivers for the Macintosh Family,* third edition. For information on the screen (`'scrn'`) resource, see *Inside Macintosh: Devices*.

## Creating File Reference, Bundle, and Signature Resources

The file reference (`'FREF'`), bundle (`'BNDL'`), and signature resources work together to give your file a distinctive appearance on the desktop. The Finder uses these resources to display the icon for your monitors extension.

The file reference resource specifies the file type for a monitors extension (`'cdev'`), the local ID of your icon list resource, and an empty string. The local ID maps the monitors extension file type to the icon list resource that is assigned the same local ID in the bundle resource. Listing 8-23 shows the file reference resource for the SurfBoard monitors extension.

**Listing 8-23** Rez input for a file reference resource of a monitors extension

```
resource 'FREF' (128, purgeable) {
   'cdev', 0, ""
};
```

**Note**

If you provide the complete icon family, the Finder displays the appropriate member of the icon family according to the bit depth of the monitor. ◆

The Finder uses the signature resource with the bundle resource to establish the identity of your monitors extension. You define a signature resource as a string resource (that is, a resource of type `'STR '`) and specify as its resource type a unique four-character sequence that has the same value as your monitors extension's creator type. The signature resource contains a string that identifies your monitors extension; typically the string specifies the name, version number, and release date of the monitors extension.

A bundle (`'BNDL'`) resource associates all of the resources that the Finder uses for your monitors extension. It associates your monitors extension and its signature with its icon. The Finder requires the information in the bundle resource to display icons for your monitors extension. In the bundle resource, you must specify a local ID for your icon list resource that matches the local ID you assigned inside the corresponding file reference resource. In the bundle resource shown in Listing 8-24, local ID 0 is assigned to the icon list resource with resource ID 128, mapping the icon defined for the SurfBoard monitors extension to the monitors extension file.

**Listing 8-24** Rez input for a bundle resource of a monitors extension

```
resource 'BNDL' (128, purgeable) {
   'kcah',
   0,
   {
   'ICN#', {0, 128},
   'FREF', {0, 128}
   }
};
```

(See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information on how to create file reference, signature, and bundle resources.)

### Including a System Extension Resource

A file that contains an extension to the Monitors control panel can contain a system extension resource of type `'INIT'`. If your monitors extension file is in the Control Panels folder, the Extensions folder, or the base level of the System Folder, then the system software executes the system extension resource when the user starts or restarts the computer.

Although the system extension resource acts independently of other resources in the file, it should be related to the monitors extension.

## Writing a Monitors Extension Function

You create a monitors extension function to implement the feature for your video card and manage the controls that allow the user to set values for that feature. The Monitors control panel calls your monitors extension function, requesting it to perform an action or handle an event in response to the user's manipulation of the controls for your video card. The `message` parameter identifies the action or event.

Your monitors extension function should perform the requested action and return a function result to the Monitors control panel. This function result should be either a standard value indicating that your monitors extension function has not allocated memory, a handle to any memory you allocate, or an error code. Here is how you declare a monitors extension function:

```
FUNCTION MyMntrExt (message, item, numItems: Integer; monitorValue: LongInt;
                    mDialog: DialogPtr; theEvent: EventRecord;
                    screenNum: Integer; VAR screens: ScrnRsrcHandle;
                    VAR scrnChanged: Boolean): LongInt;
```

The `message` parameter can contain any of the values defined by these constants:

```
CONST
   startupMsg     = 12; {status of user (whether a superuser)}
   initMsg        = 1;  {perform initialization}
   okMsg          = 2;  {user clicked OK button}
   cancelMsg      = 3;  {user clicked Cancel button}
   hitMsg         = 4;  {user clicked enabled control}
   nulMsg         = 5;  {null event}
   keyEvtMsg      = 9;  {keyboard event}
   updateMsg      = 6;  {update event}
```

The value of the `message` parameter indicates the action your monitors extension function should perform:

- `startupMsg`. Informs your monitors extension function that it has been loaded into memory. Your function can determine whether the user has superuser status by examining the `item` parameter. The Monitors control panel sets the `item` parameter to 1 if the user is a superuser. Your code should load any resources and modify them if necessary for the capabilities of the computer system or selection of superuser status. You can also allocate memory in response to this message, and store the value identifying the user's status.
- `initMsg`. Requests your monitors extension function to perform initialization.
- `okMsg`. Indicates that the user clicked the OK button. Your function should check for any values the user changed, release any memory it allocated, and return control to the Monitors control panel.
- `cancelMsg`. Indicates that the user clicked the Cancel button. Your function should restore the system to the state it was in before the user clicked the Options button, release any memory it allocated, and return control to the Monitors control panel. If the user modified any values before clicking the Cancel button, reinstate the original values.
- `hitMsg`. Indicates that the user clicked an enabled control in your monitors extension. Your function should handle the click.
- `nulMsg`. Requests your control device function to handle a null event by performing any idle processing. Your monitors extension function should do minimal processing in response to a null event; for example, it should not refresh control settings. The Monitors control panel passes the event record for the null event in the parameter `theEvent`.
- `keyEvtMsg`. Requests your monitors extension function to handle a key-down or auto-key event.
- `updateMsg`. Requests your monitors extension function to update any user items and redraw any controls that are not standard dialog items handled by the Dialog Manager.

In addition, the `message` parameter can contain any of the values defined by these constants:

```
CONST
   activateMsg    = 7;  {becoming active (not currently used)}
   deactivateMsg  = 8;  {becoming inactive (not currently used)}
   superMsg       = 10; {user is a superuser}
   normalMsg      = 11; {user is not a superuser}
```

These messages either are provided for backward compatibility or are not currently used:

- `activateMsg`. Requests your monitors extension function to respond to an activate event by making your video card's controls active. Currently, this message is not used because the Options dialog box is modal. However, your monitors extension function should handle this message as it would any activate event because in future implementations the Options dialog box might be modeless.
- `deactivateMsg`. Requests your monitors extension function to respond to an activate event by making your video card's controls inactive. Currently, this message is not used because the Options dialog box is modal. However, your monitors extension function should handle this message as it would any activate event because in future implementations the Options dialog box might be modeless.
- `superMsg`. Informs your monitors extension function that the user has selected superuser status. This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls that you have reserved for superusers, if your function has not already done so in response to either the `startupMsg` or `initMsg` message. If your function does not handle this message, it should return as its function result a handle to any memory it previously allocated. The Monitors control panel sends the message `superMsg` or `normalMsg` immediately following the initialization message.
- `normalMsg`. Informs your monitors extension function that the user has not selected superuser status. This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls, if your code has not already done so in response to either the `startupMsg` or `initMsg` message. If your function does not handle this message, it should return as its function result a handle to any memory it previously allocated. The Monitors control panel sends the message `normalMsg` or `superMsg` immediately following the initialization message.

**IMPORTANT**

If your monitors extension function cannot handle a message, it should return as its function result a handle to any memory it previously allocated. Otherwise, it should return the value passed in the `monitorValue` parameter. ▲

For a description of the remaining parameters of the monitors extension function, see "Monitors Extension Functions" beginning on page 8-78.

Your monitors extension function can return either an error code or a handle to memory it allocated. Each time the Monitors control panel calls your monitors extension function, the `monitorValue` parameter contains the value that your function returned as its function result the last time it was called.

If an error occurs, your monitors extension function should display an error dialog box and then return a value between 1 and 255. If your function returns a value in this range, the Monitors control panel closes the Options dialog box immediately and does not call your monitors extension function again.

The monitors extension used as an example in this chapter adds controls to the Options dialog box for a video card called SurfBoard. The Magnify Enabled checkbox allows the user to magnify the display of text and graphics on the monitor connected to the

SurfBoard video card. The SurfBoard monitors extension also includes controls for superusers, which illustrate how to implement the rectangle extension in which the superuser controls are displayed. The SurfBoard monitors extension shows one way of handling messages from the Monitors control panel.

Listing 8-25 shows the SurfBoard monitors extension function, `MyMonExtend`. It includes a `CASE` statement that handles messages that the Monitors control panel passes to `MyMonExtend`. First the function sets up a handle for memory that it allocates in response to the startup message. The function returns a handle to the storage it allocates as its function result in response to the startup message, unless an error occurs (see Listing 8-26 on page 8-66). For all subsequent messages, the Monitors control panel passes, in the `monitorValue` parameter, the previous function result. The `MyMonExtend` function returns the handle to the allocated memory as its function result for any messages that it does not handle.

**Listing 8-25** A monitors extension function

```
UNIT SurfBoardMonExt;
INTERFACE
   {include a Uses statement if your programming environment requires it}
CONST
   kTextItem         = 1;        {static text item}
   kSuperUserDivLine = 2;        {separation line}
   kFilterControl    = 4;        {radio button filter}
   kAntiAliasingCntl = 5;        {radio button aliasing}
   kMagnifyControl   = 6;        {checkbox for Magnify Enabled}
   kMemErrAlert      = 130;      {resource ID of out-of-memory alert box}
   kdeepAlert        = 131;      {resource ID of alert box}
   kResID            = 133;      {all other errors}
TYPE
   MonitorDataRec =
      RECORD                     {local data for the extension}
         isSuperUser:         Boolean;
         filteringSetting:    Integer;
         oldFiltering:        Integer;
         toggleMagnifyValue:  Integer;
      END;
   MonitorDataPtr       = ^MonitorDataRec;
   MonitorDataHandle    = ^MonitorDataPtr;

   MyRectHandle = ^RectPtr;
   MyIntPtr     = ^Integer;
   MyIntHandle  = ^MyIntPtr;
```

```
FUNCTION MyMonExtend (message, item, numItems: Integer;
                      monitorValue: LongInt; mDialog: DialogPtr;
                      theEvent: EventRecord; ScreenNum: Integer;
                      VAR Screens: ScrnRsrcHandle;
                      VAR ScrnChanged: Boolean): LongInt;

IMPLEMENTATION
{any support routines your monitors extension function uses}
PROCEDURE MyHandleStartupMsg(item: Integer; mDialog: DialogPtr;
                             VAR monitorValue: LongInt); FORWARD;
PROCEDURE MyHandleInitMsg(numItems: Integer; mDialog: DialogPtr;
                          dataRecHand: MonitorDataHandle); FORWARD;
PROCEDURE MyDrawRect(theWindow: WindowPtr; itemNo: Integer); FORWARD;
FUNCTION MySetUpData (superUser: Integer; storage: MonitorDataHandle): OSErr;
                      FORWARD;
PROCEDURE MyHandleHits (mDialog: DialogPtr; whichItem, numItems: Integer;
                        dataRecHand: MonitorDataHandle); FORWARD;
PROCEDURE MySaveNewValues (dataRecHand: MonitorDataHandle); FORWARD;
PROCEDURE MyUndoChanges (item, numItems: Integer; mDialog: DialogPtr;
                         dataRecHand: MonitorDataHandle); FORWARD;

FUNCTION MyMonExtend (message, item, numItems: Integer; monitorValue: LongInt;
                      mDialog: DialogPtr; theEvent: EventRecord;
                      ScreenNum: Integer; VAR Screens: ScrnRsrcHandle;
                      VAR ScrnChanged: Boolean): LongInt;
VAR
   dataRecHand:   MonitorDataHandle;
BEGIN
   IF message <> startupMsg THEN
      dataRecHand := MonitorDataHandle(monitorValue); {set up handle}
   CASE message OF
   startupMsg:
      MyHandleStartupMsg(item, mDialog, monitorValue);
   initMsg:
      MyHandleInitMsg(numItems, mDialog, dataRecHand);
   hitMsg:
      MyHandleHits(mDialog, item, numItems, dataRecHand);
   okMsg:
      MySaveNewValues(dataRecHand);
   cancelMsg:
      MyUndoChanges(item, numItems, mDialog, dataRecHand);
   END; {of CASE}
   MyMonExtend := monitorValue;{return value with handle}
END; {MyMonExtend}
```

## Handling the Startup Message

After the code in your monitors (`'mntr'`) code resource is loaded and before the Monitors control panel finds any resources to which your monitors extension function refers, the Monitors control panel calls your function with a startup (`startupMsg`) message. If the user is a superuser, the Monitors control panel sets the `item` parameter to 1 for the startup message.

The startup message requests your monitors extension function to load and modify any resources that must allow for the capabilities of the computer or for superusers. For example, your monitors extension function should modify the rectangle resource if the user is a superuser.

In response to a startup message, your function can also create a handle and allocate any memory that it needs to store values between calls from the Monitors control panel. For example, if your function initializes its controls in response to the initialization (`initMsg`) message, it should store a value indicating whether or not the user is a superuser. When the Monitors control panel calls your monitors extension function with an initialization message, the `item` parameter no longer indicates the user's status. If your code allocates memory, your function should return as its function result a handle to the memory it allocates in response to the startup message, unless an error occurs. If an error occurs, your function can display an error dialog box and return a function result of 255, indicating an error condition. Listing 8-26 shows how the `MyMonExtend` function handles the startup message.

**Listing 8-26** Handling the startup message

```
PROCEDURE MyHandleStartupMsg (item: Integer; mDialog: DialogPtr;
                              VAR monitorValue: LongInt);
VAR
   dataRecHand:   MonitorDataHandle;
   result:        OSErr;
   i:             Integer;
BEGIN
   {allocate memory to store data}
   dataRecHand :=
      MonitorDataHandle(NewHandle(sizeof(MonitorDataRec)));
   IF dataRecHand <> NIL THEN
   BEGIN
      result := MySetUpData(item, dataRecHand);
      IF result = noErr THEN
         monitorValue := LongInt(dataRecHand)
      ELSE  {error function result stops any further action}
         monitorValue := result;
   END
```

```
   ELSE
   BEGIN {dataRecHand not allocated}
      i := StopAlert(kMemErrAlert, NIL);
      {error function result stops any further action}
      monitorValue := 255;
   END;
END;
```



#### Allocating Storage in Response to the Initialization Message

If your monitors extension function does not allocate memory in response to a startup message, it can do so in response to an initialization message, and then use the superuser (`superMsg`) or the normal user (`normalMsg`) message to initialize control values and user items, if any. The Monitors control panel does not display the Options dialog box until after your monitors extension function returns from either of these messages. ◆

If your function returns an error in response to the startup message, the Monitors control panel does not display the Options dialog box. Your code can display an alert box describing the error before returning control to the Monitors control panel.

After it allocates storage, the function shown in Listing 8-26 calls its own `MySetUpData` function to check the value of the `item` parameter. This value indicates whether the user has selected superuser status.

Listing 8-27 shows the `MySetUpData` function. If the user is not a superuser, the SurfBoard monitors extension uses the default values for the rectangle resource. (This rectangle ends just before the dividing line, so that the superuser controls are not displayed.) If the user is a superuser, `MySetUpData` extends the rectangle in the rectangle (`'RECT'`) resource to include all of the controls in the item list resource (`'DITL'`) resource. If an error occurs, the function notifies the user and returns an error code value of 255 as its function result.

**Listing 8-27** Using a normal user rectangle or extending it to display superuser controls

```
FUNCTION MySetUpData(superUser: Integer; storage: MonitorDataHandle): OSErr;
VAR
   magnifyHdl:             Handle;
   intensityLevelHdl:      Handle;
   resHandle:              Handle;
   i:                      Integer;
   result:                 OSErr;
BEGIN
   result := noErr;
   HLock(Handle(storage));
   WITH storage^^ DO
   BEGIN
```

```
      {open preferences file first if needed}
      magnifyHdl := GetResource('MAGN', kResID);
      IF magnifyHdl <> NIL THEN
      BEGIN
         toggleMagnifyValue := MyIntHandle(magnifyHdl)^^;
         ReleaseResource(magnifyHdl);
      END;
      IF superUser = 1 THEN
      BEGIN
         isSuperUser := TRUE;
         intensityLevelHdl := GetResource('INTE', kResID);
         IF intensityLevelHdl <> NIL THEN
         BEGIN
            oldFiltering := MyIntHandle(intensityLevelHdl)^^;
            filteringSetting := oldFiltering;
            ReleaseResource(intensityLevelHdl);
            resHandle:= GetResource('RECT', -4096);
            IF resHandle <> NIL THEN
               RectHandle(resHandle)^^.top := -160
            ELSE
               result := 255
         END
         ELSE
            result := 255;
      END   {of superuser = 1}
      {close preferences file}
   END;  {of WITH}
   IF result = 255 THEN
   BEGIN
      DisposeHandle(Handle(storage));
      i := StopAlert(kdeepAlert, NIL);
   END;
   HUnlock(Handle(storage));
   MySetUpData := result;
END;
```

## Performing Initialization

Before it displays the Options dialog box and after it has located any resources that your monitors extension includes, such as gamma table (`'gama'`) resources, the Monitors control panel calls your monitors extension function with an `initMsg` message. When your monitors extension function receives this message, it should set default values for controls. To handle this message, your function can initialize the settings of its controls. If it hasn't already allocated memory in response to the startup message, your function can

allocate memory when it performs initialization. The Monitors control panel calls your monitors extension with an initialization message after the startup message and before either the superuser or normal message.

If your function returns an error in response to the `initMsg` message, the Monitors control panel does not display the Options dialog box. Your function can display an alert box describing the error before returning control to the Monitors control panel.

Listing 8-28 shows the `MyHandleInitMsg` procedure, which the `MyMonExtend` function calls to handle the initialization message. First `MyHandleInitMsg` sets its controls to their initial values; `MyHandleInitMsg` calls the Dialog Manager's `GetDialogItem` and the Control Manager's `SetControlValue` procedures for this purpose. Then, if the user is a superuser, the procedure installs the procedure that draws the dividing line between the normal controls and superuser controls, then initializes the settings of its superuser controls.



**Listing 8-28** Initializing a monitors extension

```
PROCEDURE MyHandleInitMsg (numItems: Integer; mDialog: DialogPtr;
                           dataRecHand: MonitorDataHandle);
VAR
   itemType:      Integer;
   itemHandle:    Handle;
   itemRect:      Rect;
BEGIN
   GetDialogItem(mDialog, numItems+kMagnifyControl, itemType,
                 itemHandle, itemRect);
   SetControlValue(ControlHandle(itemHandle),
                   (dataRecHand^^.toggleMagnifyValue));
   IF dataRecHand^^.isSuperUser THEN
   BEGIN
      GetDialogItem(mDialog, numItems+kSuperUserDivLine, itemType,
                    itemHandle, itemRect);
      SetDialogItem(mDialog, numItems+kSuperUserDivLine, itemType,
                    @MyDrawRect, itemRect);
      IF dataRecHand^^.oldFiltering = 0 THEN
         GetDialogItem(mDialog, numItems+kAntiAliasingCntl,
                       itemType, itemHandle, itemRect)
      ELSE
         GetDialogItem(mDialog, numItems+kFilterControl,
                       itemType, itemHandle, itemRect);
      SetControlValue(ControlHandle(itemHandle), 1);
   END;
END;
```

Listing 8-29 shows the `MyDrawRect` procedure, which draws the line dividing superuser controls from other controls. The `MyDrawRect` procedure uses the `FrameRect` procedure to draw a 1-pixel-high rectangle. Note that `MyDrawRect` specifies the coordinates for the dividing line in the coordinate system used by its rectangle (`'RECT'`) resource. If you wish, you can draw this line in a gray pattern so that it looks similar to the dividers in menus. (For information on the `FrameRect` procedure, see *Inside Macintosh: Imaging With QuickDraw*.)

**Listing 8-29** Drawing a line to separate superuser controls

```
PROCEDURE MyDrawRect (theWindow: WindowPtr; itemNo: Integer);
VAR
   itemType:      Integer;
   itemHdl:       Handle;
   itemRect:      Rect;
BEGIN
   GetDialogItem(theWindow, itemNo, itemType, itemHdl, itemRect);
   FrameRect(itemRect);
END;
```

## Responding to a Click in the OK Button

The Monitors control panel calls your monitors extension function with an OK (`okMsg`) message when the user clicks the OK button. The OK button is a standard control defined for the Options dialog box by the Monitors control panel. When the user clicks the OK button, the Monitors control panel hides the Options dialog box.

This message is a signal to put user preferences into effect. You should not make any changes requested by the user irreversible until you receive this message. This is your last chance to check the values of any controls or editable text items that the user might have changed. Your monitors extension function should update the resources in which it saves values; it should also make any hardware changes necessary. Your function should release any memory it has allocated before returning control to the Monitors control panel.

The `MyMonExtend` function (see Listing 8-25 on page 8-64) calls its own `MySaveNewValues` procedure to handle an OK message from the Monitors control panel. This procedure checks if the user has changed the setting of the Magnify Enabled checkbox. If the user is a superuser, it also checks the values of the Anti-Aliasing and Zirconian Filtration radio buttons. If the user changed values, `MyMonExtend` writes the values to its preferences file, which is stored in the Preferences folder, and releases any memory it has allocated before it returns to the Monitors control panel.



## Responding to a Cancel Request

When the user clicks the Cancel button, the Monitors control panel calls your monitors extension function with a cancel (`cancelMsg`) message. The Cancel button is a standard control defined for the Options dialog box by the Monitors control panel. To handle the cancel request, your monitors extension function should restore the system to its former state, before the user clicked the Options button; release any memory it allocated; and return control to the Monitors control panel. If your function modified any values the user specified before clicking the Cancel button, reinstate the original values.

## Handling Mouse Events for a Monitors Extension

When the user clicks any active enabled control that your monitors extension defined for the Options dialog box, system software generates mouse events. The Monitors control panel intercepts these events and passes them to your monitors extension function as a `hitMsg` message. Your monitors extension function typically changes the setting of the control or performs the appropriate action in response to a `hitMsg` message.

Along with the `hitMsg` message, the Monitors control panel passes three values that your monitors extension function uses to determine which item the user clicked.

- In the `item` parameter, the number of the item clicked. This is not the number you assign in your item list, but the number after the Monitors control panel appends your item list to the item list of the Options dialog box.
- In the `numItems` parameter, the number of items in the item list of the standard Options dialog box.
- In the parameter `theEvent`, the event record for the mouse event that generated the `hitMsg` message.

The Monitors control panel appends the items you define in your monitors extension item list to the item list for the standard controls in the Options dialog box. Therefore, to get the actual number of your item, subtract `numItems` from `item`.

Listing 8-30 shows the `MyHandleHits` procedure, which `MyMonExtend` calls to handle a `hitMsg` message. This procedure determines the item number of the clicked control, as defined in the monitors extension's item list resource. It does this by subtracting the number of items in the item list of the Options dialog box (`numItems`) from the item the user clicked (`whichItem`) to get the correct item number. Then `MyHandleHits` calls the Dialog Manager's `GetDialogItem` procedure and the Control Manager's `SetControlValue` procedure to set the control to the new value indicated by the user.

**Listing 8-30** Responding when a user clicks a control

```
PROCEDURE MyHandleHits (mDialog: DialogPtr; whichItem, numItems: Integer;
                        dataRecHand: MonitorDataHandle);
VAR
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
BEGIN
   HLock(Handle(dataRecHand));
   WITH dataRecHand^^ DO
   BEGIN
      CASE whichItem - numItems OF
         kFilterControl:
            BEGIN
               GetDialogItem(mDialog, whichItem, itemType, itemHandle,
                             itemRect);
               SetControlValue(ControlHandle(itemHandle),1);
               GetDialogItem(mDialog, numItems+kAntiAliasingCntl, itemType,
                             itemHandle, itemRect);
               SetControlValue(ControlHandle(itemHandle),0);
               filteringSetting := 1;
            END;
         kAntiAliasingCntl:
            BEGIN
               GetDialogItem(mDialog, numItems+kFilterControl, itemType,
                             itemHandle, itemRect);
               SetControlValue(ControlHandle(itemHandle),0);
               GetDialogItem(mDialog, whichItem, itemType, itemHandle,
                             itemRect);
               SetControlValue(ControlHandle(itemHandle),1);
               filteringSetting := 0;
            END;
```

```
            kMagnifyControl:
               BEGIN
                  GetDialogItem(mDialog, whichItem, itemType, itemHandle,
                                itemRect);
                  toggleMagnifyValue := 1 - toggleMagnifyValue;
                  SetControlValue(ControlHandle(itemHandle),
                                  toggleMagnifyValue);
               END;
         END; {end of CASE}
      END;
      HUnlock(Handle(dataRecHand));
END;
```



## Handling Keyboard Events

The Monitors control panel intercepts all key-down and auto-key events for your monitors extension and sends your monitors extension function a keyboard event through the `keyEvtMsg` message. The Monitors control panel passes, in the parameter `theEvent`, the event record for the keyboard event. If your monitors extension includes an editable text item and the user issues a Cut, Copy, or Paste command using the Command-key equivalent, the Monitors control panel passes this event to your monitors extension function in the event record.

# Including Another Control Panel Definition in a Monitors Extension File

A control panel file that contains an extension to the Monitors control panel can also contain a definition for another, separate control panel. You might want to include both an extension to the Monitors control panel and a new control panel definition in the same file, for example, if each controls some features of the same video card. Any control panel definition must include a resource of type `'cdev'` and the other resources described in “Creating a Control Panel’s Resources” beginning on page 8-14.

Because the control panel resources and the monitors extension resources in the file have different resource ID numbers, the Finder handles them separately. If the user opens a control panel file containing both a control panel definition and an extension to the Monitors control panel, the control panel defined in that file appears on the screen, and the Finder ignores the monitors extension in that file. If the user opens the Monitors control panel file, then the Monitors control panel searches the other control panel files in the same folder for extensions and ignores any control resources of type `'cdev'` it finds in those files. The user cannot open a control panel file that contains only an extension to the Monitors control panel; only the Monitors control panel can open such a file.

# Control Panels Reference

This section describes the application-defined routines and the resources that are specific to control panels and extensions to the Monitors control panel.

The section "Application-Defined Routines" describes the control device function that you must provide for a control panel and the monitors extension function that you must provide for an extension to the Monitors control panel. You create a control device function to implement a control panel. A control device function should respond to messages from the Finder, handling any events or performing any actions as requested by the Finder. A monitors extension function extends the Monitors control panel to provide support for a video device so that users can control its settings.

The "Resources" section lists the resources required for a control panel or an extension to the Monitors control panel. It includes specific sections for the resources you must supply for a control panel or a monitors extension if those resources are not fully documented elsewhere in *Inside Macintosh*, and it indicates where to find information about required resources that are not covered in that section.

## Application-Defined Routines

This section describes the control device function and the monitors extension function.

### Control Device Functions

A control device (`'cdev'`) code resource contains a control device function that implements the features of a control panel.

#### MyCdev

You provide a control device function to implement your control panel. In the `message` parameter, the Finder passes a value indicating which action your control device function should perform. Here's how you declare a control device function called `MyCdev`:

```
FUNCTION MyCdev(message, item, numItems, CPrivateValue: Integer;
                VAR theEvent: EventRecord;
                cdevStorageValue: LongInt; CPDialog: DialogPtr)
                : LongInt;
```

`message` A value that identifies the event or action to which your control device function should respond. See Table 8-3 on page 8-76 for the constants your function can receive in this parameter.

`item` The number of the item that the user clicked. In System 7, this is always the actual number of the item in your item list. In System 6, the Control Panel desk accessory appends your item list to its own. Although you begin numbering your item list with 1, the Control Panel adds the number of items in its item list to your item. Therefore, to get the actual number of the clicked item, and to provide for backward compatibility, your control device function should always subtract `numItems` from `item`.

`numItems` In System 7, the Finder passes a value of 0 for this parameter. This parameter is provided for backward compatibility with the Control Panel desk accessory. In System 6, this parameter contains the number of items in the item list belonging to the Control Panel desk accessory. To get the actual number of the item that the user clicked, subtract `numItems` from `item`.

`CPrivateValue`
Reserved for use by the Finder or the Control Panel desk accessory.

`theEvent` The event record for the event that caused the Finder to send a `hitDev`, `nulDev`, `activDev`, `deActivDev`, `updateDev`, or `keyEvtDev` message to your control device function. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of events and event records.

`cdevStorageValue`
The first time the Finder calls your control device function, this parameter is set to the constant `cdevUnset`. After the first call, this parameter contains the function result last returned by your control device function. Typically, in response to an `initDev` message, a control device function allocates a handle to memory and returns this handle as its function result. It does this so that it can store values between calls from the Finder. On all subsequent calls, the Finder passes the handle back to your function as the value of `cdevStorageValue`, and your function returns this value as its function result until an error condition occurs or the user closes the control panel.

If your function does not create a handle, your function and the Finder pass `cdevUnset` back and forth, instead of the handle, until an error condition occurs or the user closes the control panel.

`CPDialog` The dialog pointer for your control panel's dialog box. The dialog can be a color dialog on systems that support color windows. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of dialog pointers.

### DESCRIPTION

The Finder calls your control device function repeatedly with various messages in response to user actions and events from the time the user opens your control panel until the user closes the control panel or your function reports an error condition from which it cannot recover. Before attempting to handle messages from the Finder, your control device function should determine whether enough memory is available to perform the requested action.

Depending on how you define your control panel's machine resource, the Finder calls your control device function for the first time with a `macDev` message or an `initDev` message. Apart from a `macDev` message, your control device function should ignore any messages that it receives before an `initDev` message. Your function should also ignore any messages it receives after a `closeDev` message, which the Finder sends under normal conditions free of error as a signal that your function should begin its termination process: releasing any allocated memory, handles, pointers, and so on. Between the `initDev` and the `closeDev` calls, the Finder calls your control device function to direct it to handle activate, update, keyboard-related, mouse-related, and null events. When the user chooses a command from the Finder's Edit menu, the Finder passes the command to your function as an edit message. Table 8-3 lists the constant names for the values that the Finder passes in the `message` parameter and provides a description of the action your function should perform.

**Table 8-3** Messages from the Finder

| Constant | Value | Description |
|---|---|---|
| `initDev` | 0 | Your control device function should perform any initialization, set default values for controls, and create a handle to any memory that it needs. |
| `hitDev` | 1 | The user clicked an enabled item, and your control device function should handle the click. |
| `closeDev` | 2 | The user closed the control panel; your function should terminate after disposing of any handles and pointers it created. (In System 6 and earlier, the user could have also selected another control panel.) |
| `nulDev` | 3 | A null event occurred. Your control device function should perform idle processing. Do not assume any particular timing for this message. |
| `updateDev` | 4 | An update event occurred. Your control device function should update any user items and redraw any controls that are not standard dialog items handled by the Dialog Manager. |
| `activDev` | 5 | Your control panel is becoming active as the result of an activate event. Your control device function should make the default button and any other controls in your control panel active. |

**Table 8-3** Messages from the Finder (continued)

| Constant | Value | Description |
|---|---|---|
| deActivDev | 6 | Your control panel is becoming inactive as the result of an activate event. Your control device function should make the default button and any other controls in your control panel inactive. |
| keyEvtDev | 7 | A key-down or an auto-key event occurred. Your control device function should process the keyboard event. |
| macDev | 8 | Your control device function should check the hardware and software configuration to determine whether the control panel can run on it. Your function should return a function result of 1 if it can run and 0 if it cannot. |
| undoDev | 9 | The user chose the Undo command from the Finder's Edit menu. Your control device function should handle the command. |
| cutDev | 10 | The user chose the Cut command from the Finder's Edit menu. Your control device function should handle the command. |
| copyDev | 11 | The user chose the Copy command from the Finder's Edit menu. Your control device function should handle the command. |
| pasteDev | 12 | The user chose the Paste command from the Finder's Edit menu. Your control device function should handle the command. |
| clearDev | 13 | The user chose the Clear command from the Finder's Edit menu. Your control device function should handle the command. |

In System 7, the Finder processes all Command-key equivalents on behalf of your control panel, except those that it maps to its own Edit menu commands. The Finder converts these Command-key equivalents to edit messages, which it then passes to your control panel for processing. In System 6, the Control Panel passes both commands from the Edit menu and their Command-key equivalents to your control device function for processing. See the sections "Responding to Keyboard Events" on page 8-37 and "Handling Edit Menu Commands" on page 8-46 for more information on how to handle Command-key equivalents.

If your function cannot recover from an error condition, it must return one of three error codes to the Finder after disposing of any memory, handles, and pointers that it created and restoring the system stack to the state it would be in after successful execution. See Table 8-2 on page 8-47 for the error codes that your control device function can return.

**SEE ALSO**

For information on how to write a control device function, see “Writing a Control Panel Function” beginning on page 8-25. For information on the required and optional resources for your control panel, see “Creating a Control Panel’s Resources” beginning on page 8-14.

## Monitors Extension Functions

A monitor (`'mntr'`) code resource contains a monitors extension function, which adds controls to the Options dialog box of the Monitors control panel. This function implements the features that allow users to set values for the added controls.

### MyMntrExt

You provide a monitors extension function to implement the features that allow users to set the controls for your video card. Your function should respond appropriately to any messages sent to it by the Monitors control panel. In the `message` parameter, the Monitors control panel passes a value indicating which action your function should perform. Here’s how you declare a monitors extension function called `MyMntrExt`:

```
FUNCTION MyMntrExt (message, item, numItems: Integer;
                    monitorValue: LongInt; mDialog: DialogPtr;
                    theEvent: EventRecord;
                    screenNum: Integer; VAR screens: ScrnRsrcHandle;
                    VAR scrnChanged: Boolean): LongInt;
```

`message` A value that identifies the event or action to which your monitors extension function should respond. See Table 8-4 on page 8-80 for the values your function can receive in this parameter.

`item` For `hitDev` messages, the number of the item that the user clicked. The Monitors control panel appends your item list to its own. So, although you begin numbering your item list with 1 in your item list resource, the Monitors control panel adds the number of standard items in the Options dialog box’s item list to your item. Therefore, to get the actual number of the clicked item, your monitors extension function should always subtract `numItems` from `item`.

For the `startupMsg` message, the `item` parameter indicates whether the user has selected superuser status. If so, the `item` parameter is 1; if not, it is 0.

`numItems` The item list number of the last standard item in the Options dialog box.

`monitorValue`
The first time the Monitors control panel calls your monitors extension function, that is, when the `message` parameter equals `startupMsg`, the value of the `monitorValue` parameter is 0. After the first call, this

parameter contains the result your monitors extension function returned the last time the Monitors control panel called it. Because control panel routines, including a monitors extension function, cannot use global variables to store data between calls, your function can use its function result to return a handle to any memory it allocates. The next time the Monitors control panel calls your monitors extension function, it passes the handle back to your function in the `monitorValue` parameter.

If your monitors extension function returns a function result in the range 1 through 255, the Monitors control panel interprets this result as an error and closes your Options dialog box. Therefore, your monitors extension function will not receive a value in this range in the `monitorValue` parameter.

`mDialog` The dialog pointer for the Options dialog box. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of dialog pointers.

`theEvent` The event record for an event that caused the Monitors control panel to pass a `hitMsg`, `nulMsg`, or `keyEvtMsg` message to your monitors extension function. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of events and event records.

`screenNum` The number of the screen device (that is, the monitor) that the user selected. The Monitors control panel numbers monitors consecutively, in the same order as the slots in which the cards are installed, starting with 1.

`screens` A handle to a screen (`'scrn'`) resource. See *Inside Macintosh: Devices* for information on the screen resource.

`scrnChanged`

A Boolean value that you can use to indicate whether you have modified the screen (`'scrn'`) resource. Set this parameter to `TRUE` if you have modified the screen resource. When you set the `scrnChanged` parameter to `TRUE`, the Monitors control panel checks whether the values in the screen resource are still valid; if there is a problem, the Monitors control panel tries to correct it.

This parameter makes it easier to implement a control that changes the apparent area displayed on the screen. For example, your monitor might be able to display either two pages of a document or a magnified view of a single page. If the user changes the area displayed on one screen in a system with multiple screens, the displays on adjacent screens could overlap or show gaps. When you change the screen resource to implement this change, the coordinates of the global rectangles for adjacent screens are no longer contiguous. In this case, if you have set the `scrnChanged` parameter to `TRUE`, the Monitors control panel shifts the virtual locations of the screens to eliminate the gaps or overlaps.

**DESCRIPTION**

The Monitors control panel calls your monitors extension function repeatedly with messages requesting your function to perform an action or handle an event that occurs while the Options dialog box is displayed. Table 8-4 lists the constant names for the

values that the Monitors control panel passes in the `message` parameter and provides a description of the action your function should perform.

**Table 8-4** Messages from the Monitors control panel

| Constant | Value | Description |
|---|---|---|
| `initMsg` | 1 | Your monitor extension function should perform initialization; it should allocate any memory it needs and set default values for its controls.<br><br>The Monitors control panel sends this message to your function before it displays the Options dialog box but after it locates any resources, such as gamma tables, that your extension includes. |
| `okMsg` | 2 | When the user clicks the OK button, the Monitors control panel hides the Options dialog box and calls your monitors extension function with this message. This is your function's last chance to check the values of dialog items that the user might have changed. Your function should release any memory that it previously allocated before returning control to the Monitors control panel.<br><br>The OK button is a standard control put in the Options dialog box by the Monitors control panel. |
| `cancelMsg` | 3 | The user clicked the Cancel button. Your monitors extension function should return the device that your monitors extension controls to the condition it was in before the user clicked the Options button, release any memory that your function previously allocated, and return control to the Monitors control panel.<br><br>The Cancel button is a standard control put in the Options dialog box by the Monitors control panel. |
| `hitMsg` | 4 | The user clicked an enabled control in the Options dialog box, and your extension function should handle the click.<br><br>The Monitors control panel appends your item list to the standard list of items in the Options dialog box and passes, in the `item` parameter, the item's item number in the combined list. To get the actual number of the clicked item as defined in your item list, subtract `numItems` from `item`. |
| `nulMsg` | 5 | A null event occurred. Your monitors extension function should perform tasks that have to be done repeatedly, if any. Do not assume any particular timing for this message. |
| `updateMsg` | 6 | An update event occurred. Your monitors extension function should update any user items and redraw any controls that are not standard items handled by the Dialog Manager. |
| `activateMsg` | 7 | An activate event occurred, indicating that the Options dialog box is becoming active. Currently, the Monitors control panel does not call your monitors extension function with this message, because the Options dialog box is modal. However, your function should handle this message as it would any activate event, because in future versions of the Operating System the Options dialog box might be modeless. |

**Table 8-4** Messages from the Monitors control panel (continued)

| Constant | Value | Description |
|---|---|---|
| `deactivateMsg` | 8 | An activate event occurred, indicating that the Options dialog box is becoming inactive. Currently, the Monitors control panel does not call your extension function with this message, because the Options dialog box is modal. However, your function should handle this message as you would any activate event, because in future versions of the Operating System the Options dialog box might be modeless. |
| `keyEvtMsg` | 9 | A keyboard event occurred. Your monitors extension function should process the keyboard event. |
| `superMsg` | 10 | The user has selected superuser status. Your monitors extension function should display any controls that are reserved for superusers.<br><br>The Monitors control panel sends this message when the user holds down the Option key while clicking the Options button.<br><br>This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls that you have reserved for superusers, if your function has not already done this in response to either the `startupMsg` or `initMsg` message. If your code does not handle this message, it should return as its function result a handle to any previously allocated memory.<br><br>The Monitors control panel sends this message or the normal message immediately following the initialization message. |
| `normalMsg` | 11 | The user is not a superuser. This message is provided for backward compatibility with System 6. However, your monitors extension function can respond to it by initializing any controls, if your function has not already done this in response to either the `startupMsg` or `initMsg` message. If your function does not handle this message, it should return as its function result a handle to any previously allocated memory.<br><br>The Monitors control panel sends this message or the superuser message immediately following the initialization message. |
| `startupMsg` | 12 | The Monitors control panel sends this message as soon as the code in your monitors code (`'mntr'`) resource has been loaded, and before the Monitors control panel finds any resources that your monitors extension function refers to. If the user is a superuser, the Monitors control panel sets the `item` parameter to 1 when it sends the startup message.<br><br>When your monitors extension function receives this message, it can load and modify any resources that must allow for the capabilities of the system or for superusers. For example, your function can modify the item list resource to display special controls for superusers. |

Your monitors extension function can return either an error code or a value that you want to have available the next time the Monitors control panel calls your function. For example, if your monitors extension function allocates memory, it can return a handle to the memory as its function result. Each time the Monitors control panel calls your monitors extension function, the `monitorValue` parameter contains the value that your function returned the last time it was called.

Your monitors extension function must also detect and recover from any error conditions or report them to the user. If it cannot recover from an error, your monitors extension function should display an error dialog box and then return a value between 1 and 255. If your function returns a value in this range, the Monitors control panel closes the Options dialog box immediately and does not call your function again. If your function returns an error in response to the `initMsg` or `startupMsg` message, the Monitors control panel does not display the Options dialog box. Your function can display an alert box describing the error before returning control to the Monitors control panel.

**SEE ALSO**

For more information about the messages the Monitors control panel sends to your monitors extension function and how to handle them, see "Writing a Monitors Extension Function" beginning on page 8-61.

## Resources

This section identifies the resources you supply for a control panel and monitors extension. The required resources for a control panel are

- A machine (`'mach'`) resource that describes the systems on which your control panel can run or signals the Finder to call your control device function to perform this check.
- A rectangle positions (`'nrct'`) resource to define the number of rectangles that make up the control panel and their positions.
- An item list (`'DITL'`) resource to specify all of the items that are to appear in the control panel. These items can include static text, buttons, checkboxes, radio buttons, editable text, the resource IDs of icons and QuickDraw pictures, and the resource IDs of other types of controls, such as pop-up menus.
- An icon list (`'ICN#'`) resource and other icon family resources (`'ics#'`, `'icl8'`, `'icl4'`, `'ics8'`, `'ics4'`) to define the icons for the control panel file.
- A control device function (`'cdev'`) code resource that contains the code to implement the control panel.
- A file reference (`'FREF'`) resource to associate your control panels's icons with your control panel file so that the Finder can display the icons with the file type they represent.

- A bundle (`'BNDL'`) resource to associate your control panel's signature, icon list, and file reference resources.
- A signature resource—defined using a string (`'STR '`) resource—to identify your control panel.

The following required resources are described completely in chapters of *Inside Macintosh: Macintosh Toolbox Essentials* and are not included in this reference section:

- For the item list (`'DITL'`) resource, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*
- For the icon family, file reference (`'FREF'`), bundle (`'BNDL'`), and signature resources, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

The two remaining required resources—machine (`'mach'`) and rectangle positions (`'nrct'`) resources—are described in this section. The font information (`'finf'`) resource is also covered in this section; it is an optional resource that you can supply to specify the font to be used for static text items.

**Note**

You can include additional resources in your control panel file that are not required. See "Providing Additional Resources for a Control Panel" on page 8-22 for more information. ◆

The resources required for an extension to the Monitors control panel are

- A card (`'card'`) resource that contains a Pascal string identical to the name of the video card. (This is the name in the declaration ROM of the card.) Because a monitors extension can include as many card resources as you like, one extension file can handle several types of video cards.
- A monitor (`'mntr'`) code resource that contains the code to implement and handle the controls and features of your monitors extension.
- A rectangle (`'RECT'`) resource to describe the size and shape of the area used to display your controls.
- An item list (`'DITL'`) resource to specify which items you want to appear in your monitors extension. You can add additional controls for superusers, separating them from the other controls with a horizontal dividing line.

Of these required resources, the card (`'card'`), monitor (`'mntr'`), and rectangle (`'RECT'`) resources are described in this reference section. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about the item list (`'DITL'`) resource.

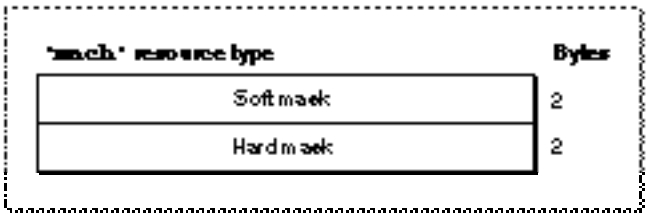
For information about the optional resources you can provide for a monitors extension, see "Supplying Optional Resources for a Monitors Extension" beginning on page 8-56.

## The Machine Resource

You can identify to the Finder the hardware and software components on which your control panel runs, or you can signal the Finder to call your control device function to perform this check. In either case, create a machine resource of type `'mach'`. A machine resource must have a resource ID of –4064.

The machine resource consists of two word-sized masks: a hard and a soft mask. Figure 8-16 shows the structure of a compiled machine resource.

**Figure 8-16** Structure of a compiled machine (`'mach'`) resource




A compiled version of a machine resource contains these elements:

- Soft mask. See Table 8-5 for a description of this mask.
- Hard mask. See Table 8-5 for a description of this mask.

The Finder performs the check if you set these masks to values representing the requirements for your control panel.

**Note**
In System 6, the Control Panel does not display the icon for a control panel file if the machine resource indicates that the control panel cannot run on the current system. ◆

If you set these masks to values indicating that the Finder is to call your control device function to perform the check, the Finder calls your function for the first time with a `macDev` message. (See “Determining If a Control Panel Can Run on the Current System” on page 8-29 for a discussion of how to handle a `macDev` message.)

Table 8-5 shows the values you use to set the machine resource masks.

**Table 8-5** Possible settings for the machine resource masks

| Soft mask | Hard mask | Action |
|---|---|---|
| `$0000` | `$FFFF` | The Finder calls this control device function with a `macDev` message, and the function must perform its own hardware and software requirements check. |
| `$3FFF` | `$0000` | This control panel runs on Macintosh II systems only. |
| `$7FFF` | `$0400` | This control panel runs on all systems with an Apple Desktop Bus (ADB). |
| `$FFFF` | `$0000` | This control panel runs on all systems. |

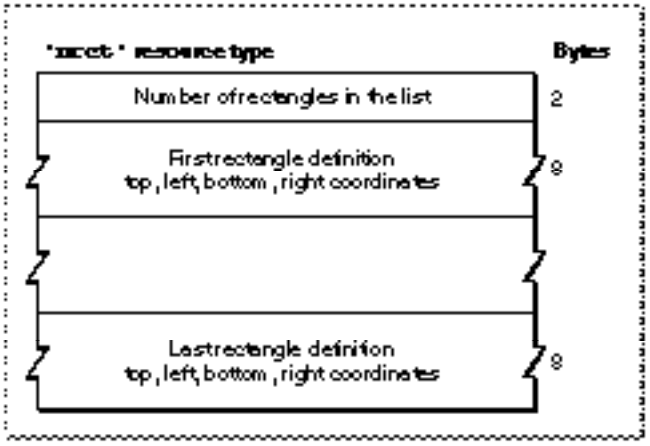
For more information about the machine resource, see “Specifying the Machine Resource” on page 8-20.



## The Rectangle Positions Resource

Your control panel can consist of one or more rectangles. To define a list of rectangles that determine the display area for your control panel, create a rectangle positions resource of type `'nrct'`. A rectangle positions resource must have a resource ID of –4064. Figure 8-17 shows the structure of a compiled rectangle positions resource.

**Figure 8-17** Structure of a compiled rectangle positions (`'nrct'`) resource



A compiled version of a rectangle positions resource contains these elements:

- Number of rectangles in the list.
- Coordinates for each rectangle. You specify the coordinates as top, left, bottom, and right.

To provide for backward compatibility with the Control Panel desk accessory, the Finder accepts only the coordinates (–1,87) as the origin of a control panel. If you are designing for System 7 only, you can extend the bottom and right edges of a control panel as far as you like. If you want your control panel to run in System 7 *and* previous versions of system software, you must limit your control panel's size to the area bounded by (–1,87,255,322). These are the coordinates used by the Control Panel desk accessory.

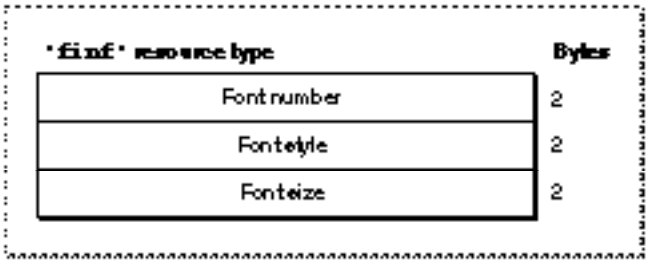
In System 6, the Control Panel desk accessory draws a frame that is 2 pixels wide around each rectangle. To join two parts of a panel neatly, overlap their rectangles by 2 pixels on the side where they meet.

For more information about the rectangle positions resource, see "Defining the Control Panel Rectangles" beginning on page 8-15.

## The Font Information Resource

The Dialog Manager uses the default application font when it displays the static text items in your control panel. To specify a different font, create a font information resource of type `'finf'`. A font information resource must have a resource ID of –4049. This is an optional resource for control panels. Figure 8-18 shows the structure of a compiled font information resource.

**Figure 8-18** Structure of a compiled font information (`'finf'`) resource




A font information resource contains three 2-byte words. A compiled version of a rectangle positions resource contains these elements:

- Font ID number. The Finder sets the graphics port's `txFont` field to this value.
- Font style. The Finder sets the graphics port's `txFace` field to this style.
- Font size. The Finder sets the graphics port's `txSize` field to this size.

For more information about the font information resource, see“Specifying the Font of Text in a Control Panel” on page 8-23.

**Note**

The Control Panel desk accessory in System 6 does not support font information resources. If your control panel can run in System 6 and you want to specify a different font, see “Defining Text in a Control Panel as User Items” on page 8-24. ◆



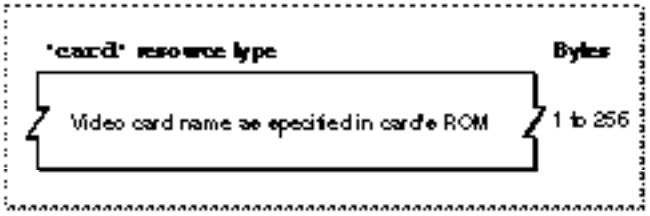
## The Control Device Function Code Resource

A control device function code resource contains the code to implement a control panel and respond to messages from the Finder. A control device function code resource is a resource of type `'cdev'` and must have a resource ID of –4064. This resource must begin with a control device function (see “Control Device Functions” beginning on page 8-74 for more information).

## The Card Resource

A card resource specifies a video card’s name. A card resource is a resource of type `'card'` and must have a resource ID within the range –4080 through –4065. A card resource contains a Pascal string—that is, a length byte followed by an ASCII string—identical to the name of a video card. The name of a video card is located in the ROM of the card, as described in *Designing Cards and Drivers for the Macintosh Family*, third edition. Figure 8-19 shows the structure of a compiled card resource.

**Figure 8-19** Structure of a compiled card (`'card'`) resource




Because a monitors extension file can contain as many card resources as you wish, one extension file can handle several types of video cards. The Options dialog box displays the name in the card resource unless you also include a string (`'STR#'`) resource in the extension file. For more information about the string resource, see “Providing an Alternative Name for a Video Card” on page 8-58.

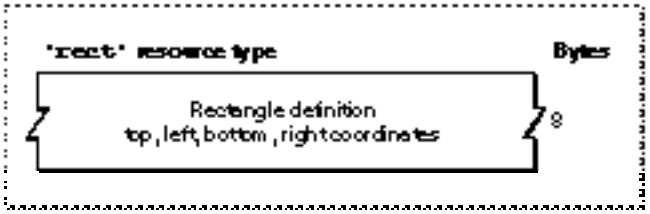
## The Monitor Code Resource

A monitor code resource contains the code that carries out the functions of a monitors extension. A monitor code resource is a resource of type `'mntr'` and must have a resource ID of –4096. This resource must begin with a monitors extension function that you provide. The Monitors control panel calls your monitors extension function with requests to perform an action or handle an event. A monitors extension should return as a function result a handle to memory that the function allocated or an error code. In MPW, you can set the code resource type to `'mntr'` when you link the program.

## The Rectangle Resource

A rectangle resource describes the display area for the controls of a monitors extension. A rectangle resource is a resource of type `'RECT'` and must have a resource ID of –4096. You specify the rectangle coordinates as top, left, bottom, and right. Figure 8-20 shows the compiled version of a rectangle positions resource.

**Figure 8-20** Structure of a compiled rectangle (`'RECT'`) resource




When enlarging the Options dialog box, the Monitors control panel places the upper edge of the new display area immediately below the lower edge of the area containing the standard controls.

When you assign coordinates to your controls, assume that the origin (that is, the upper-left corner) of the display area for your items is at (0,0). In this coordinate system, the area bounding the standard controls (such as the OK and Cancel buttons) has a right coordinate of 319 and a negative top coordinate. See “Defining a Rectangle for a Monitors Extension” on page 8-52 for an example.

Before displaying the controls defined by your monitors extension, the Monitors control panel changes the coordinates of your controls, using the coordinate system of the Options dialog box. To get the true locations of your dialog items, use the Dialog Manager’s `GetDialogItem` procedure; see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials* for information on this procedure.

# Summary of Control Panels

## Pascal Summary



### Constants

```
CONST
      {values for the message parameter for control device functions}
      initDev     = 0;  {perform initialization}
      hitDev      = 1;  {handle click in enabled item}
      closeDev    = 2;  {respond to user closing the control panel}
      nulDev      = 3;  {handle null event}
      updateDev   = 4;  {handle update event}
      activDev    = 5;  {handle activate event}
      deActivDev  = 6;  {respond to control panel becoming inactive}
      keyEvtDev   = 7;  {handle key-down or auto-key event}
      macDev      = 8;  {check whether control panel can run }
                        { on current system}
      undoDev     = 9;  {handle Undo command}
      cutDev      = 10; {handle Cut command}
      copyDev     = 11; {handle Copy command}
      pasteDev    = 12; {handle Paste command}
      clearDev    = 13; {handle Clear command}
      {initial value of cdevStorageValue}
      cdevUnset   = 3;  {the control device function has not }
                        { returned a handle}
      {error codes}
      cdevGenErr  = -1; {general error; no error dialog box is displayed }
                        { to the user}
      cdevMemErr  =  0; {not enough memory available to continue; an }
                        { out-of-memory error dialog box is displayed to }
                        { the user}
      cdevResErr  =  1; {needed resource is not available or is missing; }
                        { error dialog box is displayed to the user}

      {values for the message parameter for a monitors extension function}
      initMsg        = 1;  {perform initialization}
      okMsg          = 2;  {user clicked OK button}
```

```
cancelMsg      = 3;  {user clicked Cancel button}
hitMsg         = 4;  {user clicked enabled control}
nulMsg         = 5;  {handle null event}
updateMsg      = 6;  {handle update event}
activateMsg    = 7;  {not used}
deactivateMsg  = 8;  {not used}
keyEvtMsg      = 9;  {handle keyboard event}
superMsg       = 10; {show superuser controls}
normalMsg      = 11; {show only normal controls}
startupMsg     = 12; {gives user status (whether a superuser)}
```

## Application-Defined Routines

### Control Device Functions

```
FUNCTION MyCdev             (message, item, numItems, CPrivateValue:
                             Integer;VAR theEvent: EventRecord;
                             cdevStorageValue: LongInt;
                             CPDialog: DialogPtr): LongInt;
```

### Monitors Extension Functions

```
FUCNTION MyMntrExt          (message, item, numItems: Integer;
                             monitorValue: LongInt; mDialog: DialogPtr;
                             theEvent: EventRecord; screenNum: Integer;
                             VAR screens: ScrnRsrcHandle;
                             VAR scrnChanged: Boolean): LongInt;
```

# C Summary

## Constants

```
enum {
      /*values for the message parameter for control device functions*/
      initDev     = 0,  /*perform initialization*/
      hitDev      = 1,  /*handle click in enabled item*/
      closeDev    = 2,  /*respond to user closing control panel*/
      nulDev      = 3,  /*handle null event*/
      updateDev   = 4,  /*handle update event*/
      activDev    = 5,  /*handle activate event*/
      deActivDev  = 6,  /*respond to control panel becoming inactive*/
      keyEvtDev   = 7,  /*handle key-down or auto-key event*/
```

```
      macDev      = 8,  /*determine whether control panel can run */
                        /* on current system*/
      undoDev     = 9,  /*handle Undo command*/
      cutDev      = 10, /*handle Cut command*/
      copyDev     = 11, /*handle Copy command*/
      pasteDev    = 12, /*handle Paste command*/
      clearDev    = 13, /*handle Clear command*/
      /*initial value of cdevStorageValue*/
      cdevUnset   = 3,  /*the control device function has not */
                        /* returned a handle*/
      /*error codes*/
      cdevGenErr  = -1, /*general error; no error dialog box is displayed */
                        /* to the user*/
      cdevMemErr  = 0,  /*not enough memory available to continue; an */
                        /* out-of-memory error dialog box is displayed to */
                        /* the user*/
      cdevResErr  =  1  /*needed resource is not available or is missing; */
                        /* error dialog box is displayed */
                        /* to the user*/
   };

enum {
      /*values for the message parameter for a monitors extension*/
      initMsg        = 1,  /*perform initialization*/
      okMsg          = 2,  /*user clicked OK button*/
      cancelMsg      = 3,  /*user clicked Cancel button*/
      hitMsg         = 4,  /*user clicked enabled control*/
      nulMsg         = 5,  /*handle null event*/
      updateMsg      = 6,  /*update event*/
      activateMsg    = 7,  /*not used*/
      deactivateMsg  = 8,  /*not used*/
      keyEvtMsg      = 9,  /*handle keyboard event*/
      superMsg       = 10, /*show superuser controls*/
      normalMsg      = 11, /*show only normal controls*/
      startupMsg     = 12  /*gives user status (whether a superuser)*/
   };
```

## Application-Defined Routines

### Control Device Functions

```
pascal unsigned long MyCdev
                            (short message, short item, short numItems,
                             short CPrivateVal, const EventRecord *theEvent,
                             unsigned long cdevStorageValue,
                             DialogPtr CPDialog);
```

### Monitors Extension Functions

```
pascal unsigned long MyMntrExt
                             (short message, short item, short numItems,
                              unsigned long monitorValue,
                              DialogPtr mDialog,
                              const EventRecord *theEvent, short screenNum,
                              ScrnRsrcHandle screens, Boolean scrnChanged);
```