This chapter describes how your application can use the Dialog Manager to alert users to unusual situations and to solicit information from users. For example, in some situations your application might not be able to carry out a command normally, and in other situations the user must specify multiple parameters before your application can execute a command. For circumstances like these, the Macintosh user interface includes these two features:

- **alerts**—including alert sounds and alert boxes—which warn the user whenever an unusual or potentially undesirable situation occurs within your application
- **dialog boxes,** which allow the user to provide additional information or to modify settings before your application carries out a command

Read this chapter to learn how and when to implement alerts and dialog boxes. For example, your application can use the Dialog Manager to ask the user whether to save new or altered documents before quitting and, if the situation arises, to inform the user that there is insufficient disk space to save the file.

Virtually all applications need to implement alerts and dialog boxes. To avoid needless development effort, use the Dialog Manager to implement alerts and to create most dialog boxes. It is possible, however—and sometimes desirable—to bypass the Dialog Manager and instead use Window Manager, Control Manager, QuickDraw, and Event Manager routines to create or respond to events in complex dialog boxes. Even if you decide not to use the Dialog Manager, read this chapter for information about effective human interface design and localization issues regarding dialog boxes.

To use this chapter, you should be familiar with resources, the Event Manager, the Window Manager, and the Control Manager.

You typically use resources to specify the items you wish to display in alert boxes and dialog boxes; for example, you specify the size, location, and appearance of a dialog box in a dialog resource—a resource of type `'DLOG'`. See the chapter "Introduction to the Macintosh Toolbox" in this book for general information about resources; detailed information about the Resource Manager and its routines is provided in the chapter "Resource Manager" of *Inside Macintosh: More Macintosh Toolbox*.

The Dialog Manager offers routines that handle most of the events relating to alerts and dialog boxes, but your application still needs to handle a few additional events as described in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86. See the chapter "Event Manager" in this book for general information about events and event handling.

The Dialog Manager uses the Window Manager to display your alert boxes and dialog boxes. Although the Dialog Manager uses most of the Window Manager routines necessary to activate and update your alert and dialog boxes, your application needs to use Window Manager routines if it creates certain types of dialog boxes—such as modeless dialog boxes—as explained in this chapter. See the chapter "Window Manager" in this book for general information about windows.

The Dialog Manager uses the Control Manager to create and display buttons, radio buttons, checkboxes, and pop-up menus and to handle events in them. Generally, you shouldn't use any other controls—such as scroll bars—in your dialog boxes. If you need

to implement a more complex control, see the chapter "Control Manager" in this book. Buttons are the only controls you should use in alert boxes.

If you include editable text items in your dialog boxes, 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 brief introduction to the concepts and functions of alerts and dialog boxes, and then it discusses how you can

- create and display alerts and dialog boxes
- include controls, informative text, editable text fields, and similar items in your alert boxes and dialog boxes
- respond to events in your alert boxes and dialog boxes

Introduction to Alerts and Dialog Boxes

The behaviors and uses of alerts differ from those of dialog boxes. Important distinctions also exist between different types of alerts and between different types of dialog boxes. You choose among these according to the user's current situation.

Your application should give an alert to report an error or to issue a warning to the user. An alert can simply play a sound (called an **alert sound**) for the user, it can display an alert box that contains a message and requires an acknowledgment from the user, or it can play an alert sound and simultaneously display an alert box. **Alert boxes** are special windows that contain informative text, buttons, and, generally, icons. They may also contain pictures. As shown in Figure 6-1, an alert box typically consists of text describing why the alert appears and buttons requiring the user to acknowledge or rectify the problem.

Figure 6-1 An alert box used by the Finder

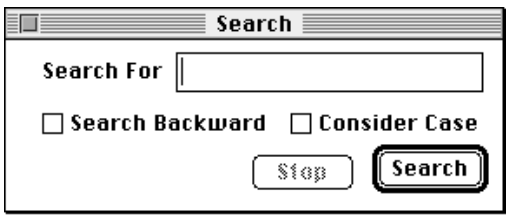
By requiring the user to click a button, an alert box obliges the user to acknowledge the alert box before proceeding. To assist the user who isn't sure how to respond when an alert box appears, your application specifies a preferred button—which invokes a preferred action—for every alert box. The Dialog Manager draws a bold outline around the preferred button so that it stands out from the other buttons in the alert box. The outlined button is also the alert box's **default button;** if the user presses the Return key

or the Enter key, the Dialog Manager acts as if the user had clicked this preferred button. For example, if the user presses the Return or Enter key in response to the alert box shown in Figure 6-1, the Dialog Manager inverts the OK button for 8 ticks and informs the Finder that the OK button has been selected; then the Finder responds by deleting the item contained in the Trash.

Use a dialog box when your application needs more information to carry out a command. Commands in menus normally act on only one object. If the user chooses a command that your application cannot perform until the user supplies more information, use a dialog box to elicit the information from the user. If a command brings up a dialog box, indicate this to your user by placing three ellipsis points (...) after the command's name in the menu.

A dialog box is a special window that typically resembles a form on which the user checks boxes and fills in blanks. Figure 6-2 shows a typical dialog box.

Figure 6-2 A typical dialog box



Although an alert typically requires only an acknowledgment to proceed from the user, a dialog box ordinarily requires the user to supply information—for instance, by entering text or by clicking a checkbox—necessary for completing the command. When you create a dialog box that carries out a command, you normally provide OK and Cancel buttons. When the user clicks the OK button, your application should perform the command according to the information that the user supplied in the dialog box. When the user clicks the Cancel button, your application should revoke the command and retract all of its actions as though the user had never given the command. Instead of using an OK button, you might use a button that describes the action to be performed; for example, you might use a Search button in a Search command's dialog box or a Remove button in a Remove command's dialog box. For simplicity, this chapter refers to the button that performs the action described in the dialog box as the *OK button*. You may even provide more than one button that performs the command, each in a slightly different way. For example, in a Change command's dialog box, you might include a Change Selection button to replace only the current selection and a Change All button to replace all occurrences throughout the entire document.

You can use any or all of the following elements in the dialog boxes you create:

- informative or instructional text
- rectangles in which text may be entered (initially blank or containing default text that can be edited)

- controls
- graphics (icons or QuickDraw pictures)
- other items as defined by your application

Types of Alerts

Every user of every application is liable to do something that the application won't understand or can't cope with in a normal manner. Alerts give your application a way to respond to these situations in a consistent manner. There are two major categories of alerts: alert sounds and alert boxes.

The **system alert sound** is a sound resource stored in the System file. This sound is played whenever system software or your application uses the Sound Manager procedure `SysBeep`. The Sound control panel allows the user to select which sound is played as the system alert sound. You can also provide your own alert sound to use in place of the system alert sound.

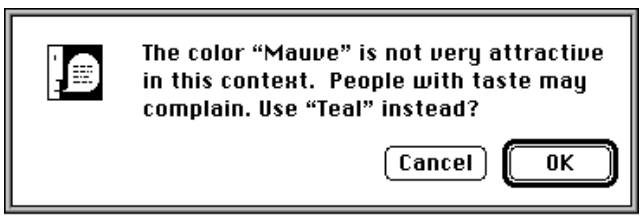
Use an alert sound for errors that are both minor and immediately obvious. For example, if the user tries to backspace past the left boundary of a text field, your application might play the alert sound instead of displaying an alert box. Your application can base its response on the number of consecutive times an alert condition recurs; the first time, your application might simply play a sound, and thereafter it might present an alert box. Your application can define different responses for each one of four alert stages.

An alert box is primarily a one-way communication from your application to the user; the only way the user can respond is by clicking buttons. Therefore, your alert boxes should contain buttons, but usually they should not contain editable text fields, radio buttons, or checkboxes—items that are typically displayed in dialog boxes.

There are three standard kinds of alert boxes: note alerts, caution alerts, and stop alerts. They are distinguished by the icons displayed in their upper-left corners.

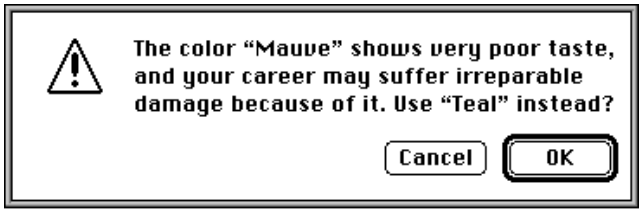
Use a **note alert** to inform users of a situation that won't have any disastrous consequences if left as is. Usually this type of alert simply offers information, and the user responds by clicking the OK button. Occasionally, as shown in Figure 6-3, a note alert may ask a simple question and provide a choice of responses.

Figure 6-3 A note alert



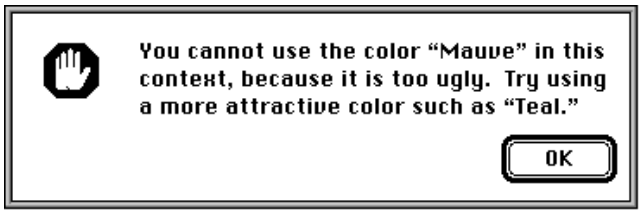
Use a **caution alert** to alert the user to an operation that may have undesirable results if it's allowed to continue. As shown in Figure 6-4, you should give the user the choice of whether to continue the action (by clicking the OK button) or to stop the action (by clicking the Cancel button).

Figure 6-4 A caution alert



Use a **stop alert** to inform the user that a problem or situation is so serious that the action cannot be completed. Stop alerts, as illustrated in Figure 6-5, typically have only a single button (OK), because all the user can do is acknowledge that the action cannot be completed.

Figure 6-5 A stop alert



You can also create **custom alert boxes** containing in the upper-left corners either your own icons or blank spaces. Plate 2 at the front of this book illustrates an alert box that the SurfWriter application displays when the user chooses the About command from the Apple menu. After reading the information in this alert box, the user clicks the OK button to dismiss it.

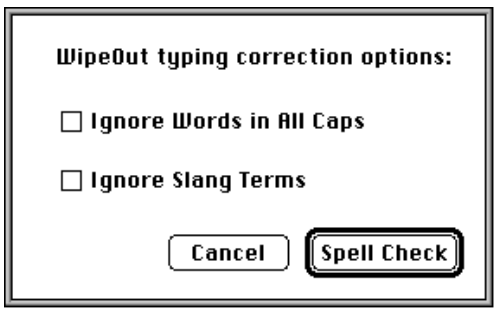
Types of Dialog Boxes

Dialog boxes should always require information from the user as well as communicate information to the user. That is, the purpose of a dialog box is to carry on a dialog between the user and your application—typically, in preparation for the execution of a command. Your dialog boxes can include editable text fields and controls such as checkboxes and radio buttons. With these, the user supplies the information your application needs to carry out the command. There are three types of dialog boxes: modal dialog boxes, movable modal dialog boxes, and modeless dialog boxes. These are described in the next three sections.

Modal Dialog Boxes

Before allowing the user to proceed with any other work, many dialog boxes require the user to click a button. The only response a user receives when clicking outside the dialog box is an alert sound. This type is called a **modal dialog box** because it puts the user in the state or "mode" of being able to work only inside the dialog box. Also called a fixed-position modal dialog box (to differentiate it from a movable modal dialog box), this type of dialog box looks like an alert box that includes other types of controls in addition to buttons. Figure 6-6 shows the modal dialog box that SurfWriter displays after the user chooses the Spell Check command.

Figure 6-6 A modal dialog box



IMPORTANT
Because the user must explicitly dismiss a modal dialog box before doing anything else, you should use a modal dialog box only when it's essential for the user to complete an operation before performing any other work. Fixed-position modal dialog boxes restrict the user's freedom of action; therefore, use them sparingly. As a rule of thumb, use a modeless dialog box whenever possible, use a movable modal dialog box whenever you can't use a modeless dialog box, and use a fixed-position modal dialog box only when you can't implement the dialog box as modeless or movable. ▲

A modal dialog box usually has at least two buttons: OK and Cancel. When the user clicks the OK button, your application should perform the command according to the information provided by the user and then remove the modal dialog box. You can give the OK button a more descriptive title if you wish. When the user clicks the Cancel button, your application should revoke any actions it took since displaying the modal dialog box, and then it should remove the modal dialog box. *Always* label this button "Cancel." Your dialog boxes can have additional buttons as well; these may or may not dismiss the dialog box.

Every dialog box you create should have a default button—that is, one whose action is invoked when the user presses the Return or Enter key. Unless you provide your own event filter function, the Dialog Manager treats the first item you specify in a description of a dialog box as the default button (that is, so long as the first item is a button). You use

an **event filter function,** described in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86, to supplement the Dialog Manager's ability to handle events; for example, an event filter function can also test for disk-inserted events and can allow background applications to receive update events. If you provide your own event filter function, it should test for key-down events involving the Return and Enter keys and respond as if the default button were clicked. The default button should invoke the preferred action, and you should try to design the preferred action to be safe—that is, so that it doesn't cause loss of data.

Although the Dialog Manager draws bold outlines around default buttons in alert boxes, it does not draw bold outlines around those in dialog boxes. To indicate the preferred action, your application should outline the default button. "Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 shows a method you can use to outline a button. If you don't outline a button in a dialog box, none should be the default button, and you must ensure in your event filter function that pressing the Return or Enter key has no effect.

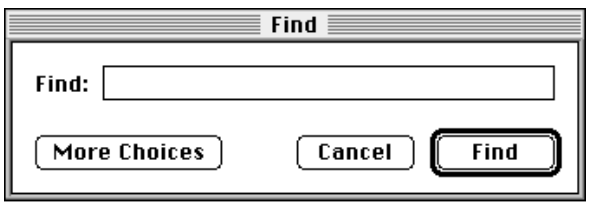
Movable Modal Dialog Boxes

The user sometimes needs to see windows obscured by an overlying modal dialog box. In this case, you should use a movable modal dialog box instead of a fixed-position modal dialog box. The **movable modal dialog box** is a modal dialog box that has a title bar so that the user can move the box by dragging its title bar.

The movable modal dialog box contains no close box and should contain no zoom box. These visual clues indicate that the user can move the dialog box, but that the dialog box is modal—that is, the user must respond to the dialog box before performing any other work in your application. If the user clicks another window belonging to your application, it should play the system alert sound. Your application removes a movable modal dialog box only after the user clicks one of its buttons. Unlike regular modal dialog boxes, however, this type of dialog box allows the user to bring another application to the front by clicking one of its windows or by choosing the application name from the Application or Apple menu.

Figure 6-7 shows the movable modal dialog box that the Finder displays after the user chooses the Find command from the File menu.

Figure 6-7 A movable modal dialog box



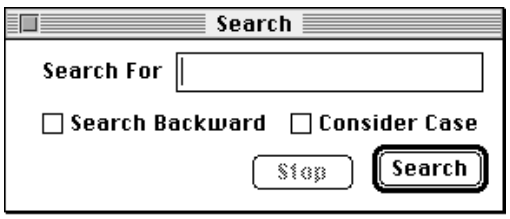
It's important to consider whether you can use a modeless dialog box instead of a modal or a movable modal dialog box—especially to preserve the user's ability to perform any task in any order.

Movable modal dialog boxes should generally respond like modal dialog boxes. Note, however, that users should be able to switch between your application and another application (thereby sending your application to the background) when you display a movable modal dialog box—an action users cannot perform with modal dialog boxes. For example, Macintosh system software uses several movable modal dialog boxes to show that the Finder is busy with a time-consuming operation (such as file copying), yet a user can still switch the Finder to the background.

Modeless Dialog Boxes

Other dialog boxes do not require the user to respond before doing anything else; these are called **modeless dialog boxes.** Whenever possible, you should try to implement your dialog boxes as modeless. As shown in Figure 6-8, a modeless dialog box looks like a document window. The user should be able to move it, make it inactive and active again, and close it like any document window. Unlike a document window, it consists mostly of buttons and other controls instead of text, and it contains no scroll bars and no size box. (A modeless dialog box should not have a size box or scroll bars; if you need these features, use the Window Manager to create a window.)

Figure 6-8 A modeless dialog box



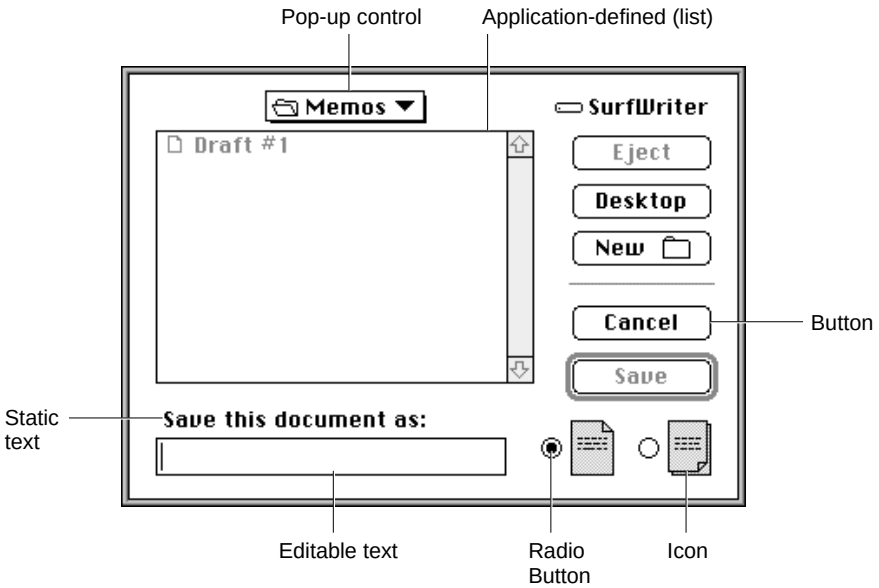
When you display a modeless dialog box, you must allow the user to perform other operations—such as working in document windows—without dismissing the dialog box. When a user clicks a button in a modeless dialog box, your application should *not* remove it; instead, the dialog box should remain on the desktop so that the user can perform the command again. Because of the difficulty in revoking the last action invoked from a modeless dialog box, it typically does not have a Cancel button, although it may have a Stop button. A Stop button in a modeless dialog box is useful for halting long printing or searching operations, for example.

When finished with a modeless dialog box, the user can click its close box or choose Close from the File menu (when the dialog box is the active window). Your application should then remove the modeless dialog box. A modeless dialog box is also dismissed implicitly when the user chooses Quit. It's usually helpful to the user for your application to remember the contents of the dialog box after it's dismissed. This way, when the user invokes the dialog box again, even after the user closes and reopens your application, you can restore the dialog box exactly as it was.

Items in Alert and Dialog Boxes

All dialog boxes and alert boxes contain items—such as icons, text, controls, and QuickDraw pictures. You use resources called **item lists** to specify which items you want to appear in your alert boxes and dialog boxes. You can even define your own items—for example, a picture whose appearance changes. Figure 6-9 illustrates most of these item types.

Figure 6-9 Typical items in a dialog box



Your application enables or disables the items it includes in its dialog and alert boxes. An **enabled item** is one for which the Dialog Manager reports user events involving that item; for example, the Dialog Manager reports to the application when a user clicks the enabled Cancel button shown in Figure 6-9. A **disabled item** is one for which the Dialog Manager does not report events. For example, the Dialog Manager does not report to the application when the user clicks or drags the static text item "Save this document as" in Figure 6-9 because that item is disabled.

Don't confuse a *disabled item* with an *inactive control*. When you don't want the Control Manager to display visual responses to mouse events in a control, you make it inactive by using the Control Manager procedure `HiliteControl`. For example, until the user types a filename, the Save button in Figure 6-9 is inactive. The Control Manager displays an inactive control in a way (such as by dimming it) that shows it's inactive. The Dialog Manager makes no visual distinction between a disabled item and an enabled item; the Dialog Manager simply doesn't inform your application when the user clicks a disabled item.

You should use `HiliteControl` to dim a control in dialog box whenever the user can't use that control. For example, Figure 6-8 shows a modeless dialog box with a dimmed

Stop button. The Stop button is dimmed because it has no effect until the user clicks the Search button. When the user initiates the search operation by clicking the Search button, the Stop button becomes active, and the Search button is dimmed.

You should use the Control Manager procedure `HiliteControl` to make the buttons and other controls inactive in a modeless or movable modal dialog box when you deactivate it. The `HiliteControl` procedure dims inactive buttons, radio buttons, checkboxes, and pop-up menus to indicate to the user that clicking these items has no effect while the dialog box is in the background. When you activate the dialog box again, use `HiliteControl` to make the controls active again.

You store information about all dialog or alert box items in an item list resource. When you use Dialog Manager routines to invoke alert boxes or create dialog boxes, the Dialog Manager gets most of the descriptive information about them from resources. The Dialog Manager calls the Resource Manager to read into memory what it needs from the resource file.

Events in Alert and Dialog Boxes

Handling events in an alert box is very simple: after you invoke an alert box, the Dialog Manager handles most events for you by automatically calling the `ModalDialog` procedure.

To handle events in a modal dialog box, your application must explicitly call the `ModalDialog` procedure after displaying the dialog box.

In either case, when an enabled item is clicked, the Dialog Manager returns the item number. You'll then do whatever is appropriate in response to that click. For mouse-down events outside the alert box or modal dialog box, the `ModalDialog` procedure plays the system alert sound and gets the next event.

The Dialog Manager automatically removes an alert box when the user clicks any enabled item. For a modal dialog box, your application should continue calling `ModalDialog` until the user selects the OK or Cancel button, and then—after responding appropriately to the user's selection—your application should remove the dialog box.

When it receives an event, `ModalDialog` passes the event to an event filter function before handling the event itself. You should provide an event filter function as a secondary event-handling loop for events that `ModalDialog` doesn't handle. For both alert and modal dialog boxes, you should provide a simple event filter function that performs the following tasks:

- return `TRUE` and the item number for the default button if the user presses the Return or Enter key
- return `TRUE` and the item number for the Cancel button if the user presses the Esc key or the Command-period key combination
- update your windows in response to update events (this also allows background applications to receive update events) and return `FALSE`
- return `FALSE` for all events that your event filter function doesn't handle

You can also use the event filter function to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor within an application-defined item.

For your application's modeless and movable modal dialog boxes, you can pass events to the `IsDialogEvent` function, or you can use your own event-handling code to learn whether the events need to be handled as part of a dialog box. If they do, call the `DialogSelect` function to assist you in handling them instead of calling the `ModalDialog` procedure. Your application should not remove a modeless dialog box unless the user clicks its close box or chooses Close from the File menu when the modeless dialog box is the active window. Your application should remove a movable modal dialog box only after the user clicks one of its enabled buttons.

Instead of using the `IsDialogEvent` or `DialogSelect` function to handle events within modeless and movable modal dialog boxes, you can use Control Manager, Window Manager, and TextEdit routines (such as `FindWindow`, `BeginUpdate`, `EndUpdate`, `FindControl`, `TrackControl`, and `TEClick`) to handle these events without the aid of the Dialog Manager.

Alert Boxes, Dialog Boxes, and the Window Manager

The Dialog Manager uses the Window Manager to draw your alert boxes and dialog boxes. You can use Window Manager or QuickDraw routines to manipulate an alert box or a dialog box just like any other window—showing it, hiding it, moving it, and resizing it.

The Dialog Manager gets most of the descriptive information about alerts and dialog boxes from resources in a resource file. An **alert resource** is a resource that describes an alert, and a **dialog resource** is a resource that describes a dialog box. Both are analogous to a window resource. (In addition to providing information that the Dialog Manager passes to the Window Manager, you also include in your alert resources and dialog resources additional information that the Dialog Manager alone uses. These resources are described more fully in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 and "Creating Dialog Boxes" beginning on page 6-23.)

When you create an alert box, the Dialog Manager always passes to the Window Manager the `dBoxProc` window definition ID for the alert box; this is so that all alert boxes have the same standard appearance and behavior. The Window Manager always displays an alert box in front of all other windows. Because an alert box requires the user to respond before doing anything else, and the response dismisses the alert box, your application typically won't need to use any Window Manager or QuickDraw routines to manipulate an alert box.

The `GetNewDialog` function for creating dialog boxes is similar to the Window Manager function `GetNewWindow`. When you call `GetNewDialog` to create a dialog box, you supply the same information as when you create a window with `GetNewWindow`. For example, you use a resource to specify the window definition ID, which determines how the dialog box looks and behaves, and a rectangle that defines the dimensions of the dialog box's graphics port. As for any window, you specify the

plane of the dialog box (which, by convention, should initially be frontmost), and you specify whether it is initially visible or invisible. If you create a dialog box that is initially invisible—for example, if you need to set a control's value before displaying it—you use the Window Manager procedure `ShowWindow` to display the dialog box.

The Dialog Manager creates the dialog window by calling the Window Manager function `NewCWindow` and then setting the window class in the window record to indicate that it's a dialog box. The Dialog Manager procedures for disposing of a dialog box, `CloseDialog` and `DisposeDialog`, are analogous to the Window Manager procedures `CloseWindow` and `DisposeWindow`.

When you create a dialog box (as described in "Creating Dialog Boxes" beginning on page 6-23), use the window definition ID of `dBoxProc` for modal dialog boxes. Use the `noGrowDocProc` window definition ID for modeless dialog boxes. (If your dialog box absolutely needs a size box or scroll bars, you should use the Window Manager to create the window instead of using the Dialog Manager.) And finally, use the `movableDBoxProc` window definition ID to create movable modal dialog boxes.

The Dialog Manager provides routines for handling most events in alert boxes and dialog boxes. For example, your application does not need to use such routines as the Window Manager function `FindWindow` and the Control Manager function `TrackControl` to determine when and where a mouse-down event occurs within an alert box's buttons. The Dialog Manager tells you which button the user clicks, and your application needs only to respond appropriately to the click. The Dialog Manager also automatically handles update and activate events for your alert boxes and dialog boxes. "Handling Events in Alert and Dialog Boxes" beginning on page 6-77 describes in detail how to use the Dialog Manager to help your application handle events.

About the Dialog Manager

The Dialog Manager greatly simplifies the task of creating alert boxes and simple modal dialog boxes. Whenever you need to create an alert box, you'll save yourself much effort by relying on the Dialog Manager. (If you need only to play the system alert sound without ever displaying an alert box for an error condition, you can use the Sound Manager procedure `SysBeep` instead of using the Dialog Manager. See *Inside Macintosh: Sound* for more information about the `SysBeep` procedure.)

You may find, however, that the advantages of using the Dialog Manager begin to diminish for dialog boxes if you make them very complex. For complex modal dialog boxes (particularly those containing multipart controls or multiple application-defined items) and for many movable modal and modeless dialog boxes, you may find it more convenient to implement your own dialog boxes using the Window Manager to create standard windows and using the Control Manager, QuickDraw, and the Event Manager to handle the tasks assumed by the Dialog Manager.

There are two main issues to consider when deciding whether to use the Dialog Manager:

- whether to use the Window Manager and the Control Manager instead of the Dialog Manager to create a dialog box

- whether to use the Event Manager, Window Manager, Control Manager, and TextEdit instead of the Dialog Manager to handle events

You may, for example, want to create complex dialog boxes by using the Dialog Manager, but then use the Event Manager, Window Manager, Control Manager, and TextEdit to handle events inside your normal event loop. With regard to movable modal and modeless dialog boxes, the sample code in this chapter illustrates such a hybrid approach: it uses the Dialog Manager to create the dialog boxes, but it uses normal event-handling code to determine an appropriate action according to which type of window is frontmost. When a modeless or movable modal dialog box is in front, this chapter illustrates how to take actions specific to that dialog box.

If you draw your own dialog box in a standard window without using the Dialog Manager, you won't be able to use Dialog Manager routines to help handle events, but in return you'll be able to update the window more quickly and extend its event handling more easily. Here are some situations that tend to diminish the advantages of using the Dialog Manager to create dialog boxes or handle events involving them:

- The dialog box contains more than 20 items.
- You need a multipart control, such as a scroll bar.
- You need to move items offscreen and onscreen.
- You need to display a moving indicator, such as a progress indicator.
- You need to display a list in the dialog box. (For more information on lists, see the chapter "List Manager" in *Inside Macintosh: More Macintosh Toolbox*.)
- You need to display text in a font other than the system font.
- Your application must respond to events other than mouse-down events, key-down events inside editable text items, and a few key-down events for keyboard equivalents when your application displays the dialog box.

If none of these situations applies to the dialog box you want to create, then you should definitely use the Dialog Manager. If only one situation applies, you should probably use the Dialog Manager. If two or more of these situations apply, you may find that it is better to create and manage a standard window that operates like a dialog box instead of using the Dialog Manager to create or manage it.

Using the Dialog Manager

You can use the Dialog Manager to

- alert users to critical situations
- carry on a dialog with users when your application needs their input

With Dialog Manager routines, you invoke alert boxes or create dialog boxes in windows whose contents are, in turn, managed by the Dialog Manager. The Dialog Manager automatically handles update events, activate events, cursor tracking, and most text-editing tasks for your alert and dialog boxes.

To implement alerts and dialog boxes, you generally

- create an alert resource or a dialog resource in a resource file
- create another resource to specify a list of items—such as controls, informative text, and pictures—to be displayed in the alert box or dialog box
- create and display the alert box or dialog box
- respond as appropriate to events relating to your alert or dialog box
- close the dialog box when you are finished with it (for alert boxes, the Dialog Manager automatically performs this for you)

These tasks are explained in greater detail in the rest of this chapter.

Before using the Dialog Manager, you must initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit, in that order. Then initialize the Dialog Manager by using the `InitDialogs` procedure.

The Dialog Manager uses the system alert sound for signaling the user during various alert stages. If you want to use alert sounds other than the system alert sound, write your own sound procedure (as illustrated in Listing 6-3 on page 6-22) and call the `ErrorSound` procedure to make it the current sound procedure.

If you want to display static text or editable text in a font other than the system font, you can use the `SetDialogFont` procedure. However, there are a number of caveats regarding this procedure. For descriptions of these caveats, see "Special Considerations" in the description of `SetDialogFont` on page 6-105.

System 7 and earlier versions of the Communications Toolbox add several new routines (namely, `AppendDITL`, `ShortenDITL`, and `CountDITL`) that make it easier for you to add items to, remove items from, and count the number of items in a dialog box. Before calling these routines, you should make sure that they are available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit field indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `AppendDITL`, `ShortenDITL`, and `CountDITL` are available.

```
CONST gestaltDITLExtPresent= 0;  {if this bit is set, then }
                                 { AppendDITL, ShortenDITL, }
                                 { & CountDITL are available}
```

The `Gestalt` function is described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities.*

Creating Alert Sounds and Alert Boxes

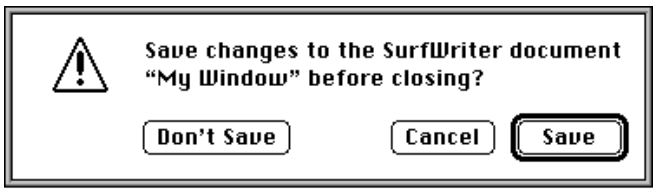
To create an alert, use one of these functions: `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert`. Icons associated with the first three functions appear in the upper-left corner of the alert boxes, as previously shown in Figure 6-3, Figure 6-4, and Figure 6-5. The `Alert` function allows you to display your own icon or to have no icon at all in the upper-left corner of the alert box.

These functions take descriptive information about the alert from an alert resource that you provide. An alert resource has the resource type `'ALRT'`. When you call one of these functions, you pass it the resource ID of the alert resource and a pointer to an event filter function. These functions create and display an alert box. When the user clicks a button in an alert box, these functions return the button's item number and close the alert box, at which time you respond appropriately to the user's click, as described in "Responding to Events in Alert Boxes" beginning on page 6-81.

Here's an example of how to create the caution alert shown in Figure 6-10.

```
VAR
   myAlertItem:      Integer;
myAlertItem := CautionAlert(kSaveAlertID, @MyEventFilter);
```

Figure 6-10 An alert box to save changes to a document



You should specify a pointer to an event filter function when you call the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. You should provide an event filter function as a secondary event-handling loop for events that `ModalDialog` doesn't handle. In this example, a pointer to `MyEventFilter` is specified for the event filter function. You can use the standard event filter function by passing `NIL` in this parameter. The standard event filter function allows users to press the Return or Enter key in lieu of clicking the default button. As described in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86, your application should provide a simple event filter function that also allows background applications to receive update events. You can use the same event filter function in most or all of your alert boxes and modal dialog boxes.

Continuing with the previous example, an application-defined constant (`kSaveAlertID`) specifies the resource ID of an alert resource in a parameter to the `CautionAlert` function. Listing 6-1 shows how this alert resource appears in Rez input format. (Rez is the resource compiler provided with Apple's Macintosh Programmer's Workshop [MPW], available from APDA.)

Listing 6-1 Rez input for an alert resource

```
resource 'ALRT' (kSaveAlertID, purgeable) {  /*alert resource*/
   {94, 80, 183, 438},        /*rectangle for alert box*/
   kSaveAlertDITL,            /*use the 'DITL' with res ID 200*/
```

```
   {                          /*alert stages, starting with #4; at each */
                              /* stage, make OK the default, display the */
                              /* alert box, & play the system alert sound*/
   OK, visible, sound1,       /*4th consecutive error*/
   OK, visible, sound1,       /*3rd consecutive error*/
   OK, visible, sound1,       /*2nd consecutive error*/
   OK, visible, sound1,       /*1st error*/
   },
   alertPositionParentWindow  /*place over document window*/
};
```

An alert resource contains the following information:

- a rectangle, given in global coordinates, that determines the alert box's dimensions and, optionally, its position; these coordinates specify the upper-left and lower-right corners of the alert box
- the resource ID of the item list for the alert box
- the actions to be taken at each of four alert stages
- as an option, a constant (either `alertPositionParentWindow`, `alertPositionMainScreen`, or `alertPositionParentWindowScreen`) that tells the Dialog Manager where to position the alert box (available only to applications running in System 7)

In Listing 6-1, the coordinates (94,80,183,438) specify the dimensions of the alert box, and the `alertPositionParentWindow` constant causes the Dialog Manager to place the alert box just below the title bar of the user's document window. If you don't supply a positioning constant, the Dialog Manager places the alert box at the global coordinates you specify for the alert box's rectangle. The positioning constants for alert boxes are explained in "Positioning Alert and Dialog Boxes" beginning on page 6-62.

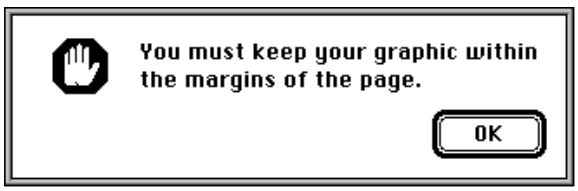
In Listing 6-1, the application-defined constant `kSaveAlertDITL` represents the resource ID for the item list resource. "Providing Items for Alert and Dialog Boxes" beginning on page 6-26 describes how to create an item list resource.

Your application can base its response on the number of consecutive times an alert condition recurs. In Listing 6-1, the alert resource specifies that each consecutive time the user repeats the action that invokes this caution alert, the Dialog Manager should perform the following: outline the OK button and treat it as the default button, display the alert box (that is, make it "visible"), and play a single system alert sound.

Your application can define different responses for each of four stages of an alert. This is most appropriate for stop alerts—those that signify that an action cannot be completed—especially when that action has a high probability of being accidental (for example, when the user chooses the Cut command when no text is selected). Under such a circumstance, your application might simply play the system alert sound the first two times the user makes the mistake, and for subsequent mistakes it might also present an alert box. Every consecutive occurrence of the mistake after the fourth alert stage is treated as a fourth-stage alert.

For example, a user might try to paste a graphic outside the page margins of a simple page-layout program; the first time the user tries this, the application—using the Dialog Manager—may simply play the system alert sound for the user. If the user repeats the mistake, the application may play the system alert sound again. But when the user repeats the error for the third consecutive time, the application may display an alert box like the one shown in Figure 6-11. If the user makes the same mistake immediately after dismissing this alert box, the alert box reappears, and it continues doing so until the user corrects or abandons the improper action.

Figure 6-11 An alert box displayed only during the third and fourth alert stages



Listing 6-2 shows the alert resource used to specify the stop alert displayed in Figure 6-11. Notice that the fourth alert stage is listed first, and the first alert stage is listed last. At the third alert stage, the application displays an alert box but does not play the system alert sound. If the user repeats the mistake a fourth consecutive time, the application plays the system alert sound and displays the alert box as well.

Listing 6-2 Specifying different alert responses according to alert stage

```
resource 'ALRT' (kStopAlertID, purgeable) {  /*alert resource*/
   {40, 40, 127, 353},     /*rectangle for alert box*/
   kStopAlertDITL,         /*use the 'DITL' with res ID 300*/
   {                       /*alert stages, starting with #4*/
   OK, visible, sound1,    /*4th err: show alert box, play alert sound*/
   OK, visible, silent,    /*3rd err: show alert box, don't play sound*/
   OK, invisible, sound1,  /*2nd err: play sound, don't show alert box*/
   OK, invisible, sound1,  /*1st err: play sound, don't show alert box*/
   },
   alertPositionParentWindow  /*place over document window*/
};
```

The actions for each alert stage are specified by the following three pieces of information:

- Which button is the default button—the OK button (that is, the first item in the item list resource) or the Cancel button (that is, the second item in the item list resource). The Dialog Manager automatically draws a bold outline around the default button, and when the user presses the Return or Enter key, the Dialog Manager treats—or your event filter function should treat—that keyboard event as a click in the default

button. The OK and Cancel buttons are described in detail in "Providing Items for Alert and Dialog Boxes" beginning on page 6-26. At each alert stage, you can change the default button, although it's difficult to imagine a scenario where changing the default button would be helpful to the user. In the previous example, the OK button is the default.

- Whether the alert box is to be displayed. If you specify the `visible` constant for an alert stage, the alert box is displayed; if you specify the `invisible` constant, it is not. In Listing 6-2, the alert box is not displayed the first two consecutive times the user repeats the mistake, but it is displayed for all subsequent consecutive times.
- Which of four possible sounds (if any) should be emitted at this stage of the alert. In the previous example, the first, second, and fourth alert stages play a single system alert sound, but the third stage plays no sound.

By default, the Dialog Manager uses the system alert sound. The `sound1` constant, used in Listing 6-2, tells the Dialog Manager to play the system alert sound once; you can also specify the `sound2` and `sound3` constants, which cause the Dialog Manager to play the system alert sound two and three times, respectively, each time at the same pitch and with the same duration. The volume of the sound depends on the current speaker volume setting, which the user can adjust in the Sound control panel. If the user has set the speaker volume to 0, the menu bar blinks once in place of each sound that the user would otherwise hear.

If you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure and then call `ErrorSound` and pass it a pointer to your sound procedure. The `ErrorSound` procedure (described on page 6-104) makes your sound procedure the current sound procedure. For example, you might create a sound procedure named `MyAlertSound`, as shown in Listing 6-3.

Listing 6-3 Creating your own sound procedure for alerts

```
PROCEDURE MyAlertSound (soundNo: Integer);
BEGIN
   CASE soundNo OF
      0: PlayMyWhisperAlert;  {sound for silent constant in alert resources}
      1: PlayMyBellAlert;     {sound for sound1 constant in alert resources}
      2: PlayMyDrumAlert;     {sound for sound2 constant in alert resources}
      3: PlayMyTrumpetAlert;  {sound for sound3 constant in alert resources}
   OTHERWISE   ;
   END;                       {of CASE}
END;
```

For each of the four alert stages that can be reported in the `soundNo` parameter, your procedure can emit any sound that you define.

As previously explained, the dimensions of the rectangle you specify in the alert resource determine the dimensions of the alert box. You can also let the rectangle coordinates serve as global coordinates that position the alert box, or you can let the Dialog Manager automatically locate it for you according to three standard positions. Listing 6-2 on page 6-21, for example, uses the `alertPositionParentWindow` constant to position the alert box over the document window where the user is working. For details about these standard positions, see "Positioning Alert and Dialog Boxes" beginning on page 6-62.

Creating Dialog Boxes

To create a dialog box, use the `GetNewDialog` or `NewDialog` function. You should usually use `GetNewDialog`, which takes information about the dialog box from a dialog (`'DLOG'`) resource in a resource file. Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages. The rest of this section describes how to use `GetNewDialog`. Although it's generally not recommended, you can also use the `NewDialog` or `NewColorDialog` function and pass it the necessary descriptive information in individual parameters instead of using a dialog resource. See page 6-118 for a description of `NewDialog` and page 6-115 for a description of `NewColorDialog`.

The `GetNewDialog` function creates a data structure (called a **dialog record**) of type `DialogRecord` from the information in the dialog resource and returns a pointer to it. A dialog record includes a window record. When you use `GetNewDialog`, the Dialog Manager sets the `windowKind` field in the window record to `dialogKind`. As explained in "Displaying Alert and Dialog Boxes" beginning on page 6-61, you can use this pointer with Window Manager or QuickDraw routines to display and manipulate the dialog box.

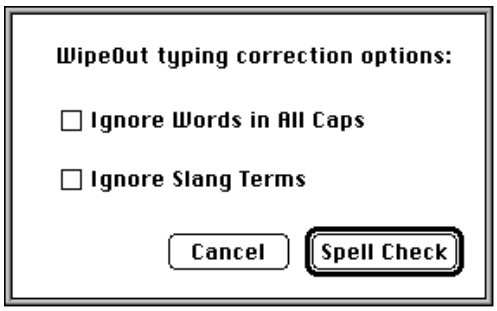
When you use `GetNewDialog`, you pass it the resource ID of the dialog resource, an optional pointer to the memory to use for the dialog record, and the window pointer `Pointer(-1)`, which causes the Window Manager to display the dialog box in front of all other windows.

If you pass `NIL` for the memory pointer, the dialog record is allocated in your application's heap. Passing `NIL` is appropriate for modal dialog boxes and movable modal dialog boxes, but—if you are creating a modeless dialog box—this can cause your heap to become fragmented. In the case of modeless dialog boxes, therefore, you should allocate your own memory as you would for a window; allocating window memory is described in the chapter "Window Manager" in this book.

Here's an example of how to create the dialog box shown in Figure 6-12.

```
VAR
   theDialog:        DialogPtr;
theDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
```

Figure 6-12 A simple modal dialog box



This example uses an application-supplied constant (`kSpellCheckID`) to specify the resource ID number of a dialog resource. Listing 6-4 shows how this dialog resource appears in Rez input format.

Listing 6-4 Rez input for a dialog resource

```
resource 'DLOG' (kSpellCheckID, purgeable) { /*dialog resource*/
   {62, 184, 216, 448},       /*rectangle for dialog box*/
   dBoxProc,                  /*window definition ID for modal dialog box*/
   visible,                   /*display this dialog box immediately*/
   noGoAway,                  /*don't draw a close box*/
   0x0,                       /*initial refCon value of 0*/
   kSpellCheckDITL,           /*use item list with res ID 400*/
   "Spellcheck Options",      /*title if this were a modeless dialog box*/
   alertPositionParentWindow  /*place over document window*/
};
```

The dialog resource contains the following information:

- a rectangle, given in global coordinates, that determines the dialog box's dimensions and, optionally, position; these coordinates specify the upper-left and lower-right corners
- the window definition ID, which specifies the window definition function and variation code for the type of dialog box
- a constant (either `visible` or `invisible`) that specifies whether the dialog box should be drawn on the screen immediately
- a constant (either `noGoAway` or `goAway`); use `goAway` only to specify a close box in the title bar of a modeless dialog box
- a reference value of type `LongInt`, which your application may use for any purpose
- the resource ID of the item list resource for the dialog box

- a text string used for the title of a modeless or movable modal dialog box
- as an option, a constant (either `alertPositionParentWindow`, `alertPositionMainScreen`, or `alertPositionParentWindowScreen`) that tells the Dialog Manager how to position the dialog box (available only to applications running in System 7)

In the example, a rectangle with coordinates (62,184,216,448) specifies the dimensions of the dialog box, and the `alertPositionParentWindow` constant causes the Dialog Manager to place the dialog box just below the title bar of the user's document window. If you don't supply a positioning constant, the Dialog Manager places the dialog box at the global coordinates you specify for the dialog box's rectangle. Positioning constants for dialog boxes are explained in "Positioning Alert and Dialog Boxes" beginning on page 6-62.

In the example, the `dBoxProc` window definition ID is used. Use the following window definition IDs for specifying dialog box types:

Window definition ID	Dialog box type
`dBoxProc`	Modal dialog box
`movableDBoxProc`	Movable modal dialog box
`noGrowDocProc`	Modeless dialog box

In each case, the Dialog Manager uses the Window Manager to draw the appropriate window frame. Figure 6-6 on page 6-10 shows an example of a modal dialog box drawn with the `dBoxProc` window definition ID, Figure 6-7 on page 6-11 shows an example of a movable modal dialog box drawn with the `movableDBoxProc` window definition ID, and Figure 6-8 on page 6-12 illustrates a modeless dialog box drawn with the `noGrowDocProc` window definition ID.

Listing 6-4 specifies the `visible` constant so that the dialog box is drawn immediately. If you use the `invisible` constant, the dialog box is not drawn until your application uses the Window Manager procedure `ShowWindow` to display the dialog box.

Use the `goAway` constant only with modeless dialog boxes. For modal dialog boxes and movable modal dialog boxes, use the `noGoAway` constant, as shown in the example.

Notice that because the example does not make use of the reference constant, 0 (`0x0`) is provided as a filler. However, you may wish to make use of this constant. For example, your application can store a number that represents a dialog box type, or it can store a handle to a record that maintains state information about the dialog box or other window types, as explained in the chapter "Window Manager" in this book. You can use the Window Manager procedure `SetWRefCon` at any time to change this value in the dialog record for a dialog box, and you can use the `GetWRefCon` function to determine its current value.

Listing 6-4 uses an application-defined constant that specifies the resource ID for the item list. The next section, "Providing Items for Alert and Dialog Boxes," describes how to create an item list.

Supply a text string in your dialog resource when you want a modeless dialog box or a movable modal dialog box to have a title. You can specify an empty string for a title bar that contains no text. The example specifies the string "Spellcheck Options" for code readability but, because the example creates a modal dialog box, no title bar is displayed.

You can let the Dialog Manager automatically locate the dialog box according to three standard positions. Listing 6-4 on page 6-24, for example, specifies the `alertPositionParentWindow` constant to position the dialog box over the document window where the user is working. For details on these standard positions, see "Positioning Alert and Dialog Boxes" beginning on page 6-62.

Providing Items for Alert and Dialog Boxes

You use an item list (`'DITL'`) resource to store information about all the items (text, controls, icons, or pictures) in an alert or dialog box. As described in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 and "Creating Dialog Boxes" beginning on page 6-23, you specify the resource ID of the item list resource in the alert (`'ALRT'`) resource or dialog (`'DLOG'`) resource.

Within an item list resource for an alert box or a dialog box, you specify its static text, buttons, and the resource IDs of icons and QuickDraw pictures. In addition, you can specify checkboxes, radio buttons, editable text, and the resource IDs of other types of controls (such as pop-up menus) in an item list resource for a dialog box.

Figure 6-13 shows an example of an alert box displayed by the SurfWriter application when the user chooses the About command from the Apple menu. To display its own icon in the upper-left corner of the alert box, the application uses the `Alert` function. An alert resource with resource ID 128 is passed in a parameter to the `Alert` function. The alert resource in turn specifies an item list resource with resource ID 128. The item list resource specifies an OK button, some static text, and an icon, whose resource ID is 128. (It's customary to assign the same resource ID to the item list resource and to either its alert resource or dialog resource, but it's not necessary to do so.)

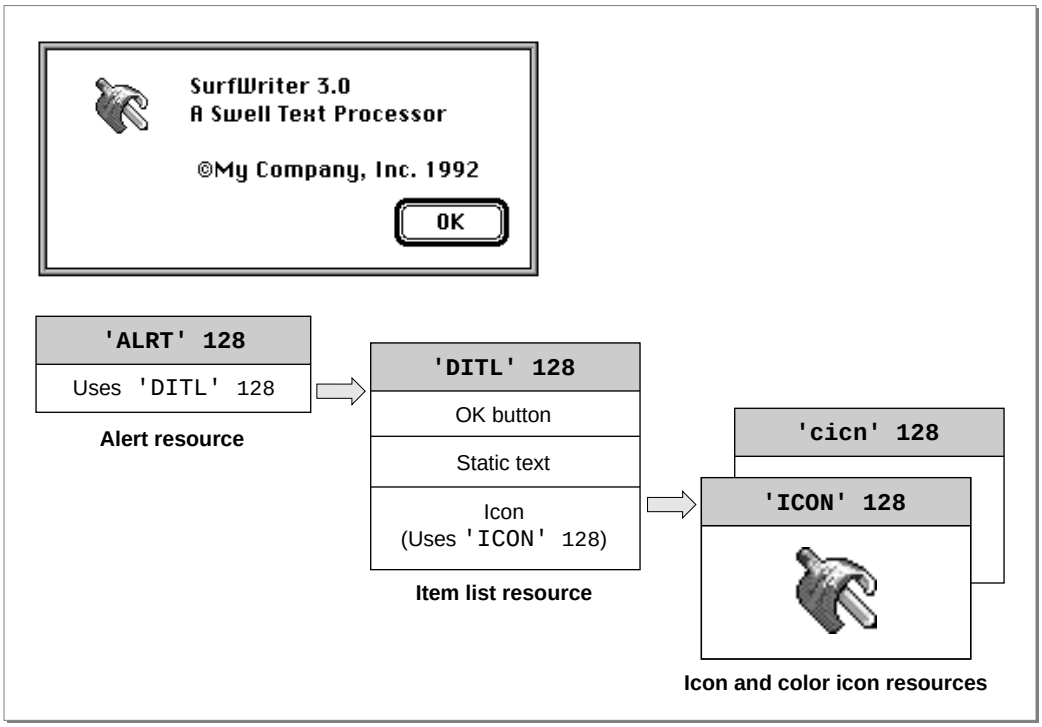
In this example, when the user chooses the About command, the SurfWriter application uses the `Alert` function to display the alert.

```
itemHit := Alert(kAboutBoxID, @MyEventFilter);
```

The `Alert` function in this example displays the alert box defined by the alert resource represented by the `kAboutBoxID` resource ID. As explained in "Responding to Events in Alert Boxes" beginning on page 6-81, the `Alert` function handles most user actions while the alert box is displayed. When the user clicks any button in an alert box, `Alert` removes the alert box and returns to your application the item number of the selected button. The application-defined function `MyEventFilter` that is pointed to in this example allows background applications to receive update events for their windows.

Listing 6-5 shows the resources, in Rez input format, that the `Alert` function uses to display the alert box shown in Figure 6-13.

Figure 6-13 Relationship of various resources to an alert box



Listing 6-5 Rez input for providing an alert box with items

```
# define kAboutBoxID 128      /*resource ID for About SurfWriter alert box*/
# define kAboutBoxDITL 128          /*resource ID for item list*/
# define kSurfWriterIconID 128      /*resource ID for 'ICON' resource*/
# define kSurfWriterColorIconID 128 /*resource ID for 'cicn' resource*/
# define kAboutBoxHelp 128          /*resource ID for 'hdlg' resource*/

resource 'ALRT' (kAboutBoxID, purgeable) {   /*About SurfWriter alert box*/
   {40, 40, 156, 309},           /*rectangle for alert box*/
   kAboutBoxDITL,                /*use item list resource with ID 128*/
   {  /*identical alert stage responses*/
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent,
      OK, visible, silent,
   },
   alertPositionMainScreen    /*display on the main screen*/
};
```

```
resource 'DITL' (kAboutBoxDITL, purgeable) { /*items for About SW alert box*/
      /*ITEM NO. 1*/
   {  {86, 201, 106, 259},    /*display rectangle for item*/
      Button {                /*the item is a button*/
            enabled,          /*enable item to return click*/
            "OK"              /*title for button is "OK"*/
      },
      /*ITEM NO. 2*/
      {10, 20, 42, 52},       /*display rectangle for item*/
      Icon {                  /*the item is an icon*/
            disabled,         /*don't return clicks on this item*/
            kSurfWriterIconID /*use 'ICON' & 'cicn' resources */
                              /* with resource IDs of 128*/
      },
      /*ITEM NO. 3*/
      {10, 78, 74, 259},      /*display rectangle for the item*/
      StaticText {            /*the item is static text*/
            disabled,         /*don't return clicks on this item*/
            "SurfWriter 3.0\n"/*text string to display*/
            "A Swell Text Processor \n\n "
            "©My Company, Inc. 1992"
      },
      /*ITEM NO. 4*/
      {0, 0, 0, 0},           /*help items get an empty rectangle*/
      HelpItem {              /*invisible item for reading in help balloons*/
            disabled,         /*don't return clicks on this item*/
            HMScanhdlg        /*scan resource type 'hdlg' for help balloons*/
            {kAboutBoxHelp}   /*get 'hdlg' with resource ID 128*/
      }
   }
};
data 'ICON' (kSurfWriterIconID, purgeable)      {
      /*icon data for black-and-white icon for About SurfWriter goes here*/
};
data 'cicn' (kSurfWriterColorIconID, purgeable)       {
      /*icon data for color icon for About SurfWriter goes here*/
};
```

Items are usually referred to by their positions in the item list resource. For example, the `Alert` function returns 1 when the user clicks the OK button in the alert box created in Listing 6-5. The Dialog Manager returns 1 because the OK button is the first item in the list. (Responding to the item numbers returned by `Alert` and other Dialog Manager functions is explained in "Handling Events in Alert and Dialog Boxes" beginning on page 6-77.) Similarly, when you use a Dialog Manager routine to manipulate an item, you specify it by its **item number,** an integer that corresponds to an item's position in its item list resource.

IMPORTANT
Item list resources should always be marked as purgeable. ▲

The Dialog Manager also calls the Resource Manager to read in any individual items as necessary.

When you create a dialog box or an alert box, the Dialog Manager creates a *dialog record.* The Dialog Manager then reads in the item list resource and stores a handle to it in the `items` field of the dialog record. Because the Dialog Manager always makes a copy of the item list resource and uses that copy, several independent dialog boxes may share the same item list resource. As explained in "Adding Items to an Existing Dialog Box" beginning on page 6-51, you can use the `AppendDITL` and `ShortenDITL` procedures to modify or customize copies of a shared item list resource for use in individual dialog boxes.

As an alternative to using a dialog resource, you can read in a dialog's item list resource directly and then pass a handle to it along with other information to `NewDialog`, which creates the dialog box. Remember, however, that it is easier to localize your application if you use a dialog resource and the `GetNewDialog` function.

An item list resource contains the following information for each item:

- a display rectangle
- the type of item (as described in the next section)
- a constant (either `enabled` or `disabled`) that instructs the Dialog Manager whether to report events for that item
- either a text string or a resource ID, as appropriate for the type of item

The **display rectangle** determines the size and location of the item. Specify the display rectangle in coordinates local to the alert or dialog box. For example, in Listing 6-5 the first item is displayed in a rectangle specified by the coordinates (86,201,106,259), which place the item in the lower-right corner of this alert box. More information about display rectangles and their placement is provided in "Display Rectangles" beginning on page 6-32.

In an item list resource, you can specify controls, text, icons, pictures, and other items that you define. In Listing 6-5, the first item's type is specified by the `Button` constant. Item types and their constants are described in the next section.

For each item in the item list resource, you must also instruct the Dialog Manager whether to report to your application when the item is clicked. In Listing 6-5, the first item is enabled, because the `enabled` constant is specified. "Enabled and Disabled Items" on page 6-36 explains when and how to enable items.

Depending on the type of item in the list, you usually provide a text string or a resource ID for the item. In Listing 6-5, the string `OK` is specified as the button title for the first item. "Resource IDs for Items" beginning on page 6-36 explains what titles and resources are appropriate for the various item types.

The information that you provide in an item list resource is described more fully in the next several sections.

Item Types

The following list shows the types of items you can include in alert and dialog boxes and the constants for specifying them in a Rez input file.

Constant	Description
`Button`	A button control
`CheckBox`	A checkbox control (use in dialog boxes only)
`Control`	A control defined in a `'CNTL'` resource file (use in dialog boxes only)
`EditText`	An editable text item (use in dialog boxes only)
`HelpItem`	An invisible item that makes the Help Manager associate help balloons with the other items defined in the item list resource
`Icon`	An icon whose black-and-white version is stored in an `'ICON'` resource and whose color version is stored in a `'cicn'` resource with the same resource ID as the `'ICON'` resource
`Picture`	A QuickDraw picture stored in a `'PICT'` resource
`RadioButton`	A radio button control (use in dialog boxes only)
`StaticText`	Static text; that is, text that cannot be edited
`UserItem`	An application-defined item (use in dialog boxes only)

The chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* describes how to create and use items of type `HelpItem` to provide help balloons for your alert and dialog boxes. When you specify a help item, make it the last item in the list, as shown in Listing 6-5 on page 6-27.

The chapter "Finder Interface" in this book describes icon (`'ICON'`) resources and color icon (`'cicn'`) resources. *Inside Macintosh: Imaging* describes `'PICT'` resources.

The chapter "Control Manager" in this book describes how to create a control with a `'CNTL'` resource. Pop-up menus are easily implemented as controls. "Pop-Up Menus as Items" beginning on page 6-42 illustrates how to include pop-up menus in your dialog boxes.

Be aware that alert boxes should contain only buttons (which the user clicks to dismiss the alert box), static text, icons, and pictures. If you need to present other items, you should create a dialog box.

The first item in an alert box's item list resource should be the OK button; if a Cancel button is necessary, it should be the second item. The Dialog Manager provides these constants for the first two item numbers:

```
CONST ok       = 1;
      cancel   = 2;
```

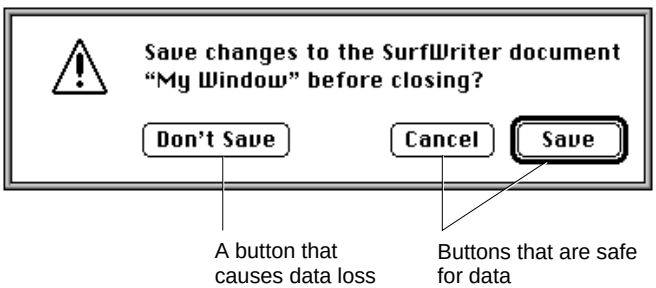
As described in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18, you define within the alert resource whether the OK or the Cancel button is the default button for each alert stage. The Dialog Manager automatically draws a bold outline around the button that you specify and, if the user presses the Return key or Enter key,

the Dialog Manager responds—or your event filter function should respond—as if the default button were clicked. (“Writing an Event Filter Function for Alert and Modal Dialog Boxes” beginning on page 6-86 describes event filter functions.)

The Dialog Manager does not draw a bold outline around the default button for dialog boxes. “Using an Application-Defined Item to Draw the Bold Outline for a Default Button” beginning on page 6-56 shows how your application can outline the default button in a dialog box. You should normally give every dialog box a default button—that is, one whose action is invoked when the user presses the Return or Enter key. “Writing an Event Filter Function for Alert and Modal Dialog Boxes” beginning on page 6-86 shows how to test for these key-down events and respond as if the user had clicked the default button. If you don’t provide your own event filter function, the Dialog Manager treats the first item in an item list resource as the default button. That is, although the Dialog Manager doesn’t draw a bold outline around the button in a dialog box, the Dialog Manager does return its item number to your application when the user presses the Return or Enter key.

Don’t set a default button to perform a dangerous action—for example, one that causes a loss of user data. If none of the possible actions is safe, don’t display a default border around any button and provide an event filter function that ignores the Return and Enter keys. This protects users from accidentally damaging their work by pressing Return or Enter out of habit. However, you should try to design a safe action that the user can invoke with a default button, such as a Save button. Figure 6-14 illustrates an alert box that provides a default button for a safe action.

Figure 6-14 A safe default button in an alert box



Provide a Cancel button whenever you can, and in your event filter function, map the Command-period key combination and the Esc (Escape) key to the Cancel button.

Don’t display a bold outline around any button if you use the Return key in editable text items, because using the same key for two different purposes confuses users and makes the interface less predictable.

Display Rectangles

As previously mentioned, the display rectangle determines the location of an item within an alert box or a dialog box. Use the alert or dialog box's local coordinates to specify the display rectangle.

For controls, the display rectangle becomes the control's enclosing rectangle. To match a control's enclosing rectangle to its display rectangle, specify an enclosing rectangle in the control resource that is identical to the one you specify for the display rectangle in the item list resource. (The control resource is described in the chapter "Control Manager" in this book.)

Note
Note that, when an item is a control defined in a control (`'CNTL'`) resource, the rectangle added to the update region is the rectangle defined in the control resource, not the display rectangle defined in the item list resource. ◆

For an editable text item, the display rectangle becomes the TextEdit destination rectangle and its view rectangle. Word wrapping occurs within display rectangles that are large enough to contain multiple lines of text, and the text is clipped if there's more than will fit in the rectangle. The Dialog Manager uses the QuickDraw procedure `FrameRect` to draw a rectangle three pixels outside the display rectangle. For more detailed information about TextEdit, see the chapter "TextEdit" in *Inside Macintosh: Text*.

For a static text item, the Dialog Manager draws the text within the display rectangle just as it draws editable text items, except that the Dialog Manager doesn't draw a frame rectangle outside the display rectangle.

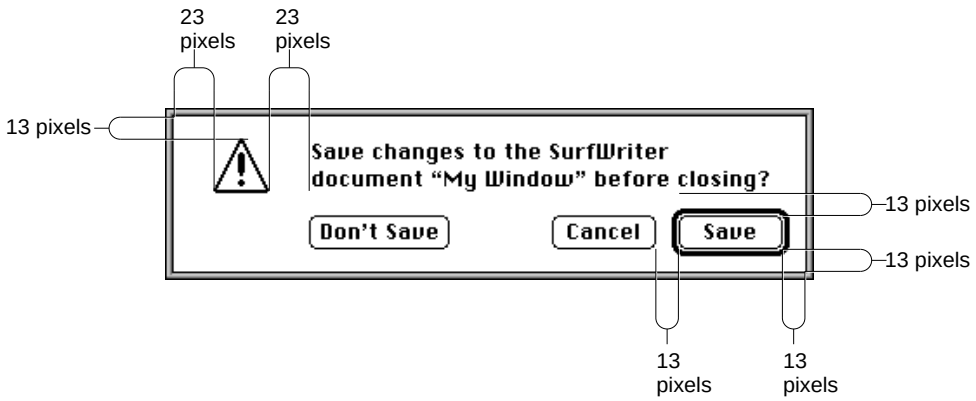
For an icon or a QuickDraw picture larger than its display rectangle, the Dialog Manager scales the icon or picture to fit the display rectangle.

Although the procedure for an application-defined item can draw outside the item's display rectangle, this is not recommended, because if the Dialog Manager receives an update event involving an area outside the display rectangle but inside the area where you draw your application-defined item, the Dialog Manager won't call your draw procedure.

Note
A click anywhere in the display rectangle is considered a click in that item. If display rectangles overlap, a click in the overlapping area is considered a click in whichever item appears first in the item list resource. ◆

You should display items in functional and consistent locations, both within your application and across all applications that you develop. In alert boxes and in most dialog boxes, place the OK button in the lower-right corner and place the Cancel button to its left. Figure 6-15 shows the recommended location of buttons and text in an alert box.

Figure 6-15 The consistent spacing of buttons and text in an alert box



Generally, you should use distances of 13 and 23 white pixels to separate the items in dialog boxes and alert boxes. (When separating the default button from the window edges and other items, don't count the pixels that make up its bold outline.) However, be aware that the Window Manager adds 3 white pixels inside the window frame when it draws alert boxes and modal dialog boxes. Therefore, specify display rectangle locations as follows when you use tools like Rez and ResEdit:

- Place the display rectangle for the lower-right button 10 pixels from the right edge and 10 pixels from the bottom edge of the alert or modal dialog box; align the display rectangles for other bottommost and rightmost items with this button.
- Place the display rectangle for the upper-left icon (or similar item) 10 pixels from the top edge and 20 pixels from the left of the alert or modal dialog box; align the display rectangles for other topmost and leftmost items with this item. The Dialog Manager automatically places the caution, note, and stop alert icons in this position when you use the `CautionAlert`, `NoteAlert`, and `StopAlert` functions. When you use the `Alert` function, you must specify the icon and the location.
- Place the other elements in the alert or modal dialog box 13 or 23 pixels apart, as shown in Figure 6-15.

For example, the rectangle for the alert box in Figure 6-15 has a specified height of 85 pixels. The display rectangle for the Save button has a bottom coordinate of 75, and the display rectangle for the static text item has a top coordinate of 10. The Window Manager adds 3 white pixels at the top of the alert box and 3 pixels at the bottom, so the alert box contains 13 white pixels below the Save button and 13 white pixels above the static text display rectangle. Listing 6-6 shows how the locations for these display rectangles are specified in a Rez input file.

Listing 6-6 Rez input for consistent spacing of display rectangles

```
resource 'DITL' (200, purgeable) {
   {  {55, 288, 75, 348},  Button      {enabled, "Save"},
      {55, 215, 75, 275},  Button      {enabled, "Cancel"},
      {55, 72,  75, 156},  Button      {enabled, "Don't Save"},
      {10, 75,  42, 348},  StaticText  {disabled,
         "Save changes to the SurfWriter document “^0” before"
         " closing?"}
   }
};
```

When specifying display rectangle locations for items in movable modal and modeless dialog boxes, use the full distance of either 13 or 23 pixels to separate items from the window edges. For example, if the items in Figure 6-15 were placed in a modeless dialog box, the top coordinate of the Save button's display rectangle should be 52 instead of 55, and its bottom coordinate would be 72 instead of 75.

As explained in the previous section, the default button can be any button; its assignment is secondary to the consistent placement of buttons. This rule keeps the OK button and the Cancel button consistently placed. Otherwise, the buttons would keep changing location depending on the default choice.

The Western reader's eye tends to move from the upper-left area of the alert or dialog box to the lower-right area. For Western versions of your software, use the upper-left area for elements (such as the alert icon) that convey the initial impression that you want to make. Place the buttons that a user clicks in the lower-right area.

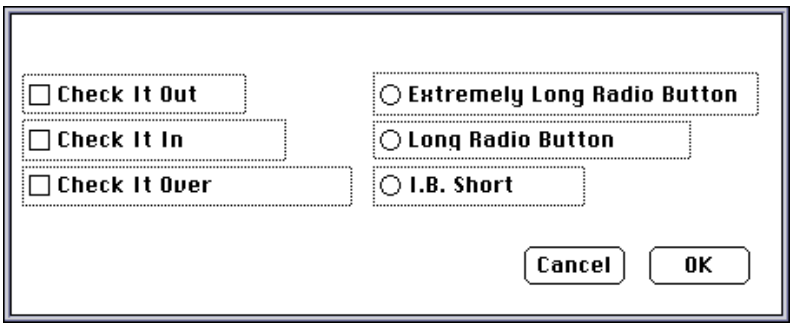
The alignment of the items in an alert box or a dialog box may vary with localization. Although in Roman script systems these items are generally aligned left to right, items in Arabic or Hebrew script systems should generally be aligned right to left, because Arabic and Hebrew are written from right to left. The TextEdit procedure `TESetJust`, described in the chapter "TextEdit" in *Inside Macintosh: Text*, controls the alignment of interface elements.

When line alignment is right to left, as in Hebrew and Arabic, the Control Manager transposes checkboxes—and radio buttons—and their titles. That is, checkboxes and radio buttons appear to the right of the text instead of to the left, as in Roman script systems. Therefore, when you create checkboxes, radio buttons, and static text items that need to align, make sure that their display rectangles are the same size.

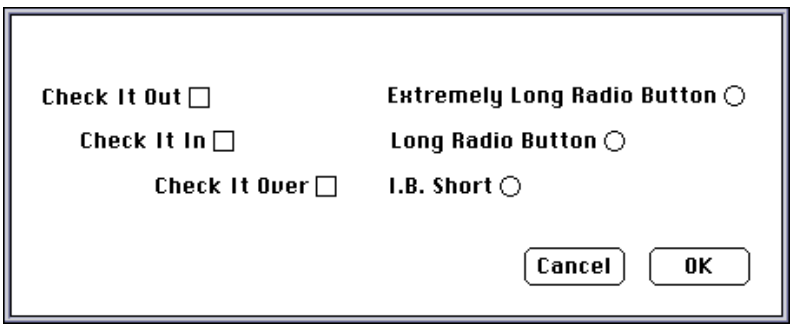
The dialog box at the top of Figure 6-16 shows several checkboxes and radio buttons with display rectangles of different sizes. The next dialog box in the figure illustrates what happens to the alignment of these items after the Control Manager transposes the controls with their titles.

The bottom two dialog boxes in Figure 6-16 illustrate how the Control Manager displays properly sized items when transposing the controls with their titles.

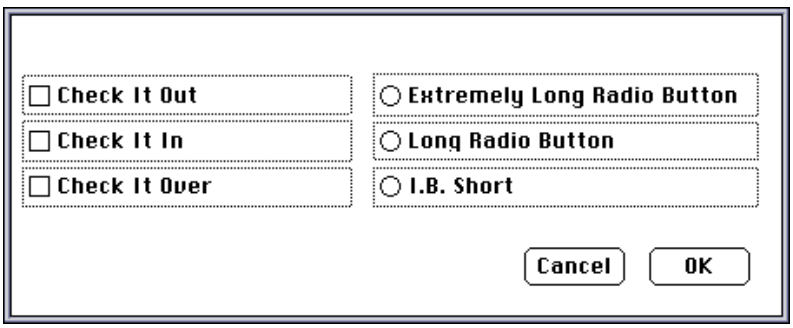
Figure 6-16 Incorrectly and correctly sized display rectangles for alternate script systems



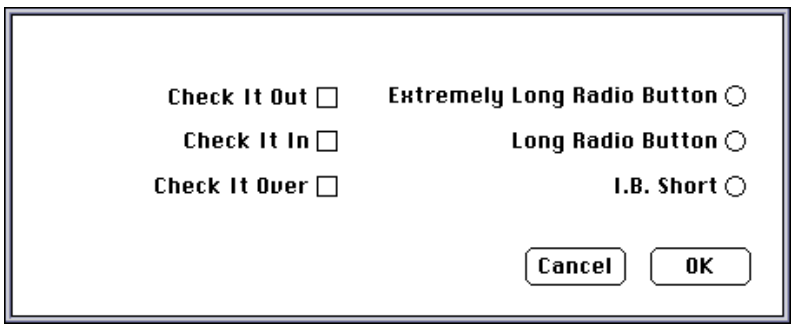
A dialog box containing display rectangles of different sizes



The same dialog box after the Control Manager transposes checkboxes and radio buttons with their titles



A dialog box containing display rectangles of the same sizes



The same dialog box after the Control Manager transposes checkboxes and radio buttons with their titles

Enabled and Disabled Items

For each item in an item list resource, include one of these two constants to specify in a Rez input file whether the Dialog Manager should inform your application of events involving that item:

Constant	Description
`enabled`	Informs your application about events involving this item
`disabled`	Doesn't inform your application about events involving this item

Generally, you should enable only controls. In particular, you should enable buttons, radio buttons, and checkboxes so that your application knows when they've been clicked. You typically disable editable text and static text items. You normally disable editable text items because you use the Dialog Manager function `GetDialogItemText` to read the information in the items only after the user clicks the OK button. (Listing 6-12 on page 6-49 illustrates how to use the `GetDialogItemText` function for this purpose.) You should use static text items only for providing information; users don't expect to click them. Likewise, you typically disable icons and pictures that merely provide information; if you use an icon or a picture as a buttonlike control to receive input, however, you must enable it. If you create an application-defined item such as a moving indicator to display information, you typically disable it. If you create an application-defined item such as a buttonlike control to receive input, you must enable it.

Don't confuse disabling an item with using the Control Manager procedure `HiliteControl` to make a control inactive. When you don't want the Control Manager to respond to clicks in a control, you make it *inactive*; when you don't want the Dialog Manager to report clicks in a control, you make it *disabled*.

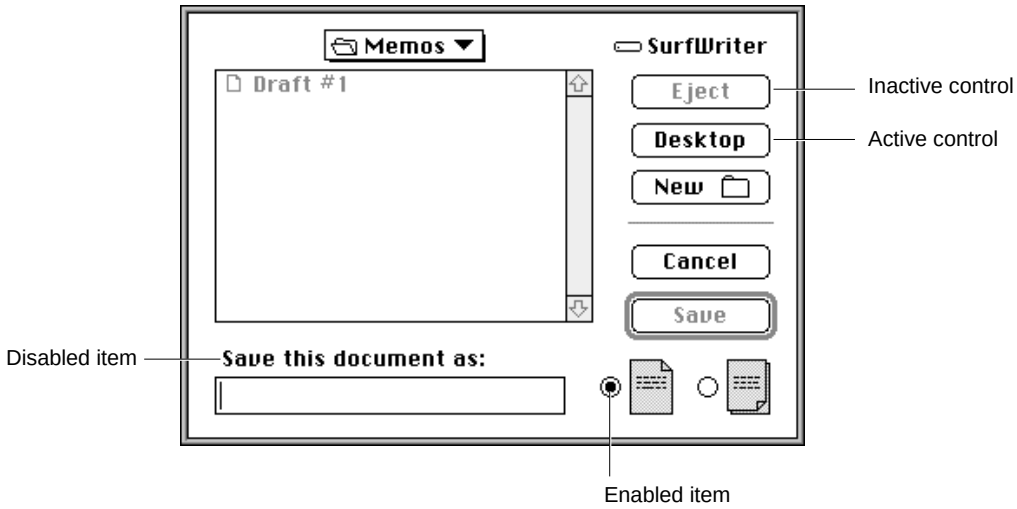
The Control Manager displays an inactive control in a way (dimmed, for example) that shows it's inactive, whereas the Dialog Manager makes no visual distinction between a disabled item and an enabled item. Figure 6-17 shows the difference between an inactive and an active control. The Control Manager procedure `HiliteControl` has been used to dim the inactive Eject button. If a user clicks this button, the Control Manager does not respond. However, when a user clicks the active Desktop button, the Control Manager inverts the button for 8 ticks.

Buttons and other controls are generally enabled, and disabling them does not alter their appearance; the enabled radio button in Figure 6-17 would appear the same if it were disabled. Because the static text reading "Save this document as" in Figure 6-17 is not a control, the application doesn't need to respond clicks in the text. Therefore, the application has disabled it; however, the static text would have the same appearance if the application were to enable it.

Resource IDs for Items

The final element for an item in an item list resource is usually either a text string or a resource ID. The choice depends on the type of item.

Figure 6-17 Inactive controls and disabled items



Provide a resource ID for icons, QuickDraw pictures, and controls other than buttons, checkboxes, and radio buttons. For an icon, provide the ID of an `'ICON'` resource; for a QuickDraw picture, the ID of a `'PICT'` resource; and for a control (including a pop-up menu), the ID of a `'CNTL'` resource. In Listing 6-5 on page 6-27, the resource ID of 128 specifies which `'ICON'` (and `'cicn'`) resources to use for the second item in the item list resource.

For a button, checkbox, radio button, static text item, and editable text item, supply a text string as the final element for the item in its item list resource. The next several sections provide guidelines for the text that you should provide.

For your own application-defined items, supply neither a title nor a resource ID. Listing 6-15 on page 6-57 shows an item list resource that includes an application-defined item.

Titles for Buttons, Checkboxes, and Radio Buttons

For a button, checkbox, or radio button, provide a text string for the control's title as the final element for the item when you specify it in the item list. In Listing 6-5 on page 6-27, the string `OK` specifies the button title for the first item in the item list resource.

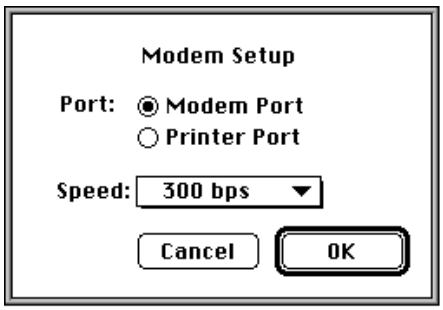
Use book-title capitalization for these items. In general, this means that you capitalize one-word titles and, in multiple-word titles, words of four or more letters. Usually you don't capitalize words such as *in*, *an*, or *and*. The rules for capitalization of titles appear in the *Apple Publications Style Guide*, which is available from APDA.

As explained in the chapter "Control Manager" in this book, the title of a checkbox should reflect two clearly opposite states, because a checkbox should allow the user to turn a particular setting either on or off. The opposites should be expressed in an equal sense in either state. If you can't devise a checkbox title that clearly implies its opposite state, you might be better off using radio buttons. With radio buttons, you can use two

titles, thereby clarifying the states. Radio buttons should represent related, but not necessarily opposite, choices. Give each radio button a title consisting of a word or a phrase that identifies what the button does. Remember that, as described in the chapter "Control Manager" in this book, the radio buttons in a group are mutually exclusive: only one button in that group can be on at one time.

Whenever possible, title a button with a verb describing the action that the button performs. A user typically reads the text in an alert box or a dialog box until it becomes familiar and then relies on visual cues, such as button titles or positions, to respond. Buttons such as Save, Quit, or Erase Disk allow users to identify and click the correct button quickly. These titles are often more clear and precise than OK, Yes, and No. If the action can't be condensed into a word or two, OK and Cancel or Yes and No may serve the purpose. If you use these generic titles, be sure to phrase the wording in the dialog box so that the action the button initiates is clear. Figure 6-18 shows a dialog box with appropriate OK and Cancel buttons.

Figure 6-18 A dialog box with OK and Cancel buttons



Cancel means "dismiss this operation with no side effects." It does not mean "I've read this dialog box" or "stop what you're doing regardless." When users click the Cancel button in your alert boxes, modal dialog boxes, and movable modal dialog boxes, your application should revoke any actions it took since displaying the alert or dialog box and then remove the box.

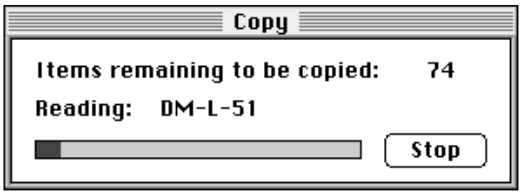
Your application should not remove a modeless dialog box when the user clicks a button; rather, you should remove the dialog box when the user clicks its close box or chooses Close from the File menu while the modeless dialog box is active.

When it is impossible to return to the state that existed before an operation began, don't use a Cancel button. You can use Stop or OK, which are useful in different situations. A Stop button may leave the results of a partially complete task intact, whereas a Cancel button always returns the application and its documents to their previous state. Use OK for a button that closes the alert box, modal dialog box, or movable modal dialog box and accepts any changes made while the dialog box was displayed.

Because of the difficulty in revoking the last action invoked from a modeless dialog box, these dialog boxes typically don't have Cancel buttons, although they may have Stop buttons. For example, the movable modal dialog box shown in Figure 6-19 uses a Stop

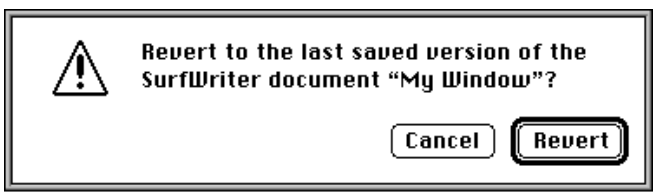
button; clicking the button halts the current file copy operation but leaves intact the copies that were previously made.

Figure 6-19 A movable modal dialog box with a Stop button



In an alert box that requires confirmation, use a button title that describes the result of accepting the message in the alert box. For example, if an alert box asks "Revert to the last saved version of the document," use a Revert button rather than an OK button. Try to use a verb in the button title; as in the Revert button in Figure 6-20, use the same verb that you use in your alert message.

Figure 6-20 An alert box with a Revert button



If the alert box presents the user with a situation in which no alternative actions are available, give the box a single button that's titled OK. You should interpret the user's clicking this button to mean "I've read the alert box."

A modal dialog box usually cuts the user off from the task. That is, when making choices in a modal dialog box, the user can't see the area of the document that changes. The user sees the changes only after dismissing the dialog box. If the changes aren't appropriate, then the user has to repeat the entire operation. To provide better feedback to the user, you need to give the user a way to see what the changes will be. Therefore, any selection made in a modal dialog box should immediately update the document contents, or you should provide a sample area in the dialog box that reflects how the user's selections will change the document. In the case of immediate document updating, the OK button means "accept this change" and the Cancel button means "undo all changes made through this dialog box."

The Dialog Manager displays button titles (as well as all other control titles) in the system font. To make it easier to localize your application, you should not change the font.

Text Strings for Static Text and Editable Text Items

For an editable text item, if you want the item to display only a blinking cursor, specify an empty string as the item's final element in the item list resource or specify a string if you want to display some default text.

For a static text item, supply a text string as its final element when you specify it in the item list resource. In the third item in Listing 6-5, the text string `SurfWriter 3.0 \nA Swell Text Processor \n\n©My Company, Inc. 1992` specifies the alert box's message.

Whenever you provide static text items in alert and dialog boxes, ensure that the messages make sense to the user. Use simple, nontechnical language and don't provide system-oriented information to which the user can't respond.

Whenever applicable, state the name of the document or application in your alert or dialog box. For example, the alert box in Figure 6-20 on page 6-39 shows both the name of the application (SurfWriter) and the name of the document (My Window). This kind of message helps users who are working with several documents or applications at once to make decisions about each one individually. "Changing Static Text" beginning on page 6-46 describes how to use the `ParamText` procedure to supply the names of document windows to your alert and dialog boxes dynamically.

Use icons and pictures whenever possible. Images can describe some error situations better than words, and familiar icons help users distinguish their alternatives better. However, because experience has shown that it is nearly impossible to create icons that are comprehensible or inoffensive across all international markets, you should be prepared to localize any icons or pictures you use. See the chapter "Icons" in *Macintosh Human Interface Guidelines* for more information about creating appropriate icons.

For your static text items, it's generally better to be polite than abrupt, even if it means lengthening your message. Your message should be helpful, and it may offer constructive suggestions, but it should not appear to give orders. Its focus should be to help the user perform the task, not to give an interesting but academic description of the task itself.

When you localize your application for use with other languages, the text may become longer or shorter. Translated text is often 50 percent longer than U.S. English text. You may need to resize your display rectangles and your alert and dialog boxes to accommodate the translated text.

By default, the Dialog Manager displays static text items in the system font. To make it easier to localize your application, you should not change the font. Do not use a smaller font, such as 9-point Geneva; some script systems such as KanjiTalk require 12-point fonts. You will save yourself future localization effort by leaving all the text in your alert and dialog boxes in the system script.

In alert boxes, try to include information that tells the user how to resolve the problem at hand. Never refer the user to external documentation for further clarification.

Stop alerts typically report errors to the user. A good error message explains what went wrong, why it went wrong, and what the user can do about it. Express this information in the user's vocabulary, not in your programming vocabulary.

Figure 6-21 shows an example of a very poor alert message—the information is expressed in the programmer's vocabulary, and the user is offered no clue about how to remedy the problem.

Figure 6-21 An obscure and useless alert message

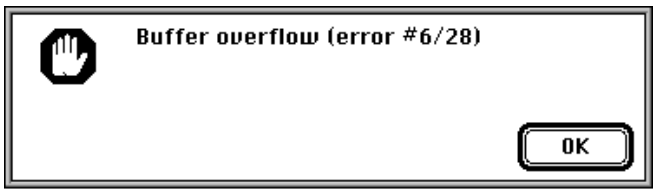


Figure 6-22 shows a somewhat better alert message. Although the vocabulary is less technical, no remedy to the problem is offered.

Figure 6-22 A less obscure alert message

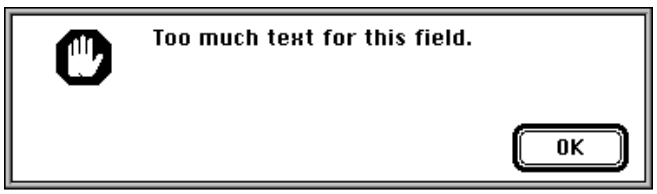
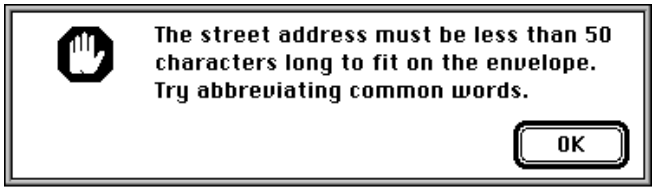


Figure 6-23 illustrates a good alert message. The message is specific, it's expressed in nontechnical terms, it explains why the error occurred, and it suggests a solution to the problem.

Figure 6-23 A clear and helpful alert message

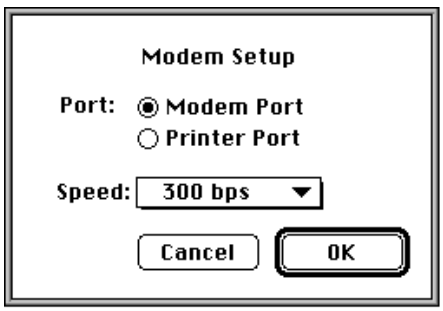


The best way to make an alert message understandable is to think carefully through the error condition itself. Can the application handle this without an error? Is the message specific enough so that the user can fix the situation? What are the recommended solutions?

Pop-Up Menus as Items

You can use pop-up menus to present the user with a list of mutually exclusive choices in a dialog box. Figure 6-24 illustrates a typical use of pop-up menus in a dialog box. As explained in the chapter "Control Manager" in this book, pop-up menus are especially useful as an alternative to radio buttons when the user must make one choice from a list of many or set a specific value, or when you must present a variable list of choices. The pop-up menu in Figure 6-24 allows the application to present a choice of modem speeds that vary according to the modem type in the user's computer.

Figure 6-24 A pop-up menu in a dialog box



In System 7, pop-up menus are implemented as controls. To display a pop-up menu in a dialog box, you

- define specific features of the pop-up menu in a control that uses the standard pop-up control definition function (described in the chapter "Control Manager" in this book)
- define the menu items of a pop-up menu just as you define items in other menus (using `GetMenu` or `NewMenu`, as described in the chapter "Menu Manager" in this book)
- specify the pop-up menu in the dialog box's item list resource

Using the pop-up control definition function, the Dialog Manager automatically draws the pop-up box and its drop shadow, inserts the text into the pop-up box, draws a downward-pointing triangle, and draws the pop-up menu's title. When the user moves the cursor to a pop-up menu and presses the mouse button, the pop-up control definition function highlights the pop-up menu title, displays the pop-up menu, and handles all user interaction until the user releases the mouse button. When the user releases the mouse button, the pop-up control definition function closes the pop-up box, draws the user's choice in the pop-up box (or restores the previous item if the user doesn't make a new choice), and removes the highlighting from the pop-up menu title. The control definition function then sets the value of the control to the item selected by the user. Your application can use the Control Manager function `GetControlValue` to get the number of the currently selected item.

The modal dialog box shown in Figure 6-24 is created by defining a dialog resource that describes the dialog box and by defining an item list resource that describes the dialog items, including a control whose `'CNTL'` resource uses the standard pop-up control definition function. Listing 6-7 shows the dialog resource and item list resource for this modal dialog box.

Listing 6-7 Rez input for a dialog resource and an item list resource for a dialog box that includes a pop-up menu

```
resource 'DLOG' (kModemDialog, purgeable) {
   {62, 184, 216, 416}, dBoxProc, visible, noGoAway, 0x0,
   kModemDialogDITL, "", alertPositionMainScreen
};
resource 'DITL' (kModemDialogDITL, purgeable) {
   {  {123, 152, 144, 222},   Button      {enabled, "OK"},
      {123, 69, 144, 139},    Button      {enabled, "Cancel"},
      {13, 70, 33, 204},      StaticText  {enabled, "Modem Setup"},
      {41, 23, 61, 64},       StaticText  {enabled, "Port:"},
      {41, 67, 59, 186},      RadioButton {enabled, "Modem Port"},
      {59, 67, 77, 186},      RadioButton {enabled, "Printer Port"},
      {90,18,109,198},        Control     {disabled, kPopUpCNTL},
      {123, 152, 144, 222},   UserItem    {disabled}  /*outline OK button*/
      {0,0,0,0},              HelpItem    {disabled, HMScanhdlg{kModemHelp}}
                                                      /*Balloon Help*/
   }
};
```

Listing 6-8 shows the `'CNTL'` and `'MENU'` resources for the Speed pop-up menu shown in Figure 6-24. Notice that the display rectangle specified for the control in the item list resource is the same as the enclosing rectangle specified in the control resource. See the chapter "Control Manager" in this book for a complete description of how to specify values for a pop-up menu's control resource.

Listing 6-8 Rez input for a control resource and a menu resource for a pop-up menu

```
resource 'CNTL' (kPopUpCNTL, preload, purgeable) {
   {90, 18, 109, 198},  /*enclosing rectangle of control*/
   popupTitleLeftJust,  /*title position*/
   visible,             /*make control visible*/
   50,                  /*pixel width of title*/
   kPopUpMenu,          /*'MENU' resource ID*/
   popupMenuCDEFProc,   /*pop-up control definition ID*/
   0,                   /*reference value*/
   "Speed:"             /*control title*/
};
```

```
resource 'MENU' (kPopUpMenu, preload, purgeable) {
   mPopUp, textMenuProc,
   0b11111111111111111111111111111111,
   enabled, "Speed",
   {
      "300 bps",     noicon, nokey, nomark, plain;
      "1200 bps",    noicon, nokey, nomark, plain;
      "2400 bps",    noicon, nokey, nomark, plain;
      "9600 bps",    noicon, nokey, nomark, plain;
      "19200 bps",   noicon, nokey, nomark, plain
   }
};
```

Keyboard Navigation Among Items

Your dialog boxes may have several items, such as editable text items and scrolling lists, that can accept input from the keyboard. You need to give users a visual cue indicating which item is currently accepting input from the keyboard. Each item type has its own distinct indicator. The Dialog Manager automatically displays a blinking cursor in an editable text item to indicate that it is accepting keyboard input. You can also use the `SelectDialogItemText` procedure (explained on page 6-131) to indicate a selected text range within an editable text item.

When a scrolling list is accepting keyboard input, you should indicate it by a rectangular border of two black pixels, separated from the list by one pixel of white space. In Figure 6-25, the AppleTalk Zones scrolling list is the item currently accepting keyboard input in the Chooser dialog box. See the chapter "List Manager" in *Inside Macintosh: More Macintosh Toolbox* for details about creating lists in dialog boxes.

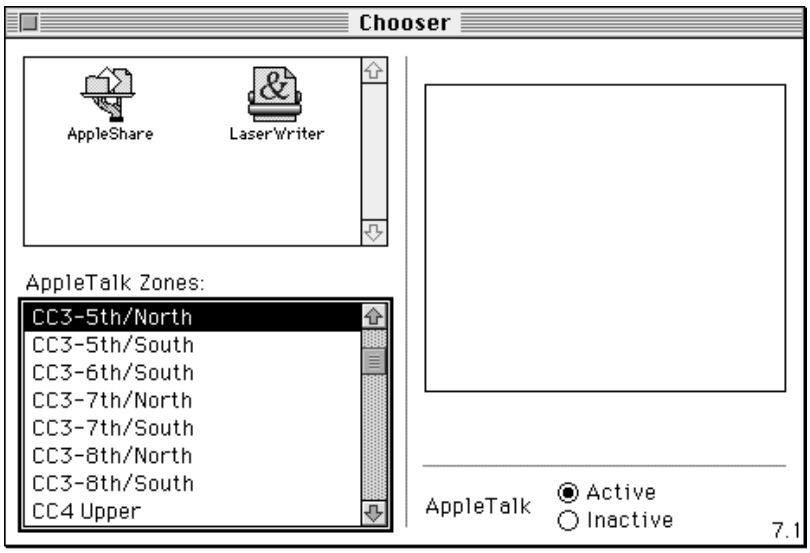
Because all typing goes to the active window, there should be only one active area and only one indicator at any time. If only one element in a dialog box can accept keyboard input and that element is a scrolling list, it's not necessary to place a border around it.

The Dialog Manager automatically handles mouse-down events and keyboard events for the Tab key. Thus, the user can select any item that accepts keyboard input by clicking the desired item or by pressing the Tab key to cycle through the available items. When the user presses the Tab key, the Dialog Manager accepts the changes made to the current item and selects the next item—as listed in the item list—that accepts keyboard input. When the user clicks another item, the Dialog Manager accepts the changes made to the current item and selects the newly clicked item.

Manipulating Items

In many cases, you won't have to make any changes to alerts or dialog boxes after you define them in your resource file. However, if you should want to modify an item, you can use several Dialog Manager routines to do so.

Figure 6-25 A selected scrolling list



For example, you can use the `ParamText` procedure to supply text strings (such as document titles) to alert and dialog boxes dynamically. For most other types of item manipulation, you must first call the `GetDialogItem` procedure to get the information about the item. You then use other routines to manipulate that item. For example, you can use the `SetDialogItem` procedure to change the item, or—to get a text string that the user has entered in an editable text item after clicking the OK button—you can use the `GetDialogItemText` procedure.

The Dialog Manager routines for manipulating items are summarized in the following list.

Routine	Description
`AppendDITL`	Adds items to a dialog box.
`CountDITL`	Counts the number of items in a dialog box.
`FindDialogItem`	Finds an item that contains a specified point within a dialog box.
`GetAlertStage`	Returns the stage of the last occurrence of an alert.
`GetDialogItem`	Returns the item type, the display rectangle, and the control handle or application-defined procedure of a given item in a dialog box.
`GetDialogItemText`	Returns the text of a given editable text or static text item.
`HideDialogItem`	Hides the given item.
`ParamText`	Substitutes up to four different text strings in static text items.
`ResetAlertStage`	Resets the stage of the last occurrence of an alert.

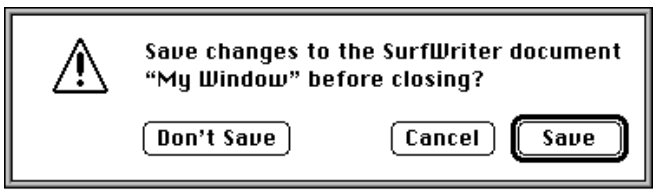
Routine	Description
`SelectDialogItemText`	Selects the text of an editable text item.
`SetDialogItem`	Sets the item type and the display rectangle of an item, or (for application-defined items) the draw procedure of an item.
`ShortenDITL`	Removes items from a dialog box.
`ShowDialogItem`	Redisplays the item previously hidden by `HideDialogItem`.

The next several sections describe the most frequently used of these routines. The next section, "Changing Static Text," explains the use of the `ParamText` procedure to manipulate the text in static text items. "Getting Text From Editable Text Items" beginning on page 6-48 describes how to use the `GetDialogItemText` procedure to determine what the user types in an editable text item. Using the `AppendDITL` procedure is explained in "Adding Items to an Existing Dialog Box" beginning on page 6-51. "Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 describes how to use `SetDialogItem` to install application-defined items. For additional information about all of the previously listed routines, see "Manipulating Items in Alert and Dialog Boxes" beginning on page 6-120 and "Handling Text in Alert and Dialog Boxes" beginning on page 6-129.

Changing Static Text

As previously explained, it is often useful to state the name of a document in an alert box or a dialog box. For example, Figure 6-26 shows an alert box that an application might display when the user closes a window that contains unsaved changes.

Figure 6-26 An alert box that displays a document name



You can use the `ParamText` procedure to supply the names of document windows to your alert and dialog boxes dynamically, as illustrated in the application-defined routine `MyCloseDocument` shown in Listing 6-9.

Listing 6-9 Using the `ParamText` procedure to substitute text strings

```
PROCEDURE MyCloseDocument (myData: MyDocRecHnd);
VAR
   title:      Str255;
   item:       Integer;
   docWindow:  WindowPtr;
   event:      EventRecord;   {dummy parameter for calling DialogSelect}
   myErr:      OSErr;
BEGIN
   docWindow := FrontWindow;     {point to active window}
   IF (myData^^.windowDirty) THEN   {document has been changed}
   BEGIN
      GetWTitle(docWindow, title);  {get title of window}
      MyStringCheck(title);
      ParamText(title, '', '', ''); {pass the title in 1st parameter}
      DoActivate(docWindow, FALSE, event);   {deactivate the active window}
      item := CautionAlert(kSaveAlertID, @MyEventFilter); {display alert box}
      IF item = kCancel THEN
         Exit(MyCloseDocument);
      IF item = kSave THEN
         DoSaveCmd;        {save the document}
      myErr := DoCloseFile(myData);    {close the file}
   END;                 {let click in Don't Save fall through}
   CloseWindow(docWindow);
   DisposPtr(Ptr(docWindow));
END;
```

In this example, the Window Manager function `FrontWindow` returns a pointer to the active window. Another Window Manager function, `GetWTitle`, returns the title of that window. The `MyCloseDocument` routine passes this string to the `ParamText` procedure, which takes four text strings as parameters. In this example, only one string is needed (the window title), which is passed in the first parameter; empty strings are passed for the remaining three parameters.

You can use `ParamText` to supply up to four text strings for a single alert or dialog box. In the item list resource for the alert or dialog box, specify where each of these strings should go by inserting the special characters `^0` through `^3` in any of the items where you can specify text. The `ParamText` procedure dynamically replaces `^0` with the string you pass in its first parameter, `^1` with the string in the second parameter, and so forth, when you display the alert or dialog box.

IMPORTANT

To avoid recursion problems in versions of system sofware earlier than 7.1, you have to ensure that you do not include the characters `^0` through `^3` in any strings you pass to `ParamText`. This is why `MyCloseDocument` uses another application-defined routine, `MyStringCheck`, to filter these characters out of the window titles passed to `ParamText`. ▲

Listing 6-10 shows a portion of an item list resource. When the application calls `CautionAlert`, the Dialog Manager uses the first parameter passed previously to the `ParamText` procedure to replace the characters `^0` in the static text with the title of the document window.

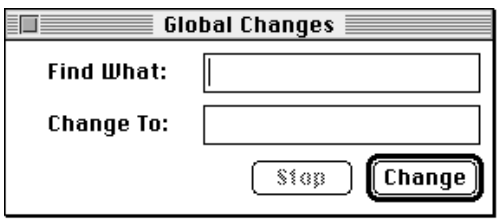
Listing 6-10 Specifying where `ParamText` should substitute text in an alert box message

```
resource 'DITL' (kSaveAlertID, purgeable) {
   {        /*Save button information goes here*/
            /*Cancel button information goes here*/
            /*Don't Save button information goes here*/
      {10, 75, 42, 348},
      StaticText {      /*ask the user to save changes to the document--*/
            disabled,   /* filename inserted with ParamText*/
            "Save changes to the SurfWriter document “^0” before closing?"
      },
            /*help item information goes here*/
   }
};
```

Getting Text From Editable Text Items

The application displaying the modeless dialog box shown in Figure 6-27 uses the `GetDialogItem` and `GetDialogItemText` procedures after the user clicks the Change button.

Figure 6-27 Two editable text items in a modeless dialog box



This dialog box prompts the user for two text strings: one to search for and another to take the place of the first string. Listing 6-11 shows the item list resource for this dialog box. The fifth item in the list is the editable text item where the user enters the text string being sought; the sixth item is the item where the user enters the replacement text string.

Listing 6-11 Specifying editable text items in an item list

```
resource 'DITL' (kGlobalChangesDITL, purgeable) {
   {  /*ITEM NO. 1*/
      {70, 213, 90, 271},  Button      {enabled,"Change"},
      /*ITEM NO. 2*/
      {70, 142, 90, 200},  Button      {enabled,"Stop"},
      /*ITEM NO. 3*/
      {10, 23, 27, 98},    StaticText  {disabled, "Find What:"},
      /*ITEM NO. 4*/
      {40, 23, 57, 98},    StaticText  {disabled,"Change To:"},
      /*ITEM NO. 5*/
      {10, 117, 27, 271},  EditText    {disabled, ""},
      /*ITEM NO. 6*/
      {40, 117, 57, 271},  EditText    {disabled, ""}
      /*ITEM NO. 7: for drawing outline around Change button*/
      {63, 205, 97, 278},  UserItem    {disabled, },
      /*ITEM NO. 8: help item goes here*/
   }
};
```

Listing 6-12 shows how the application handles a click in the Change button. (Subsequent sections of this chapter explain how to handle events in a modeless dialog box.)

Listing 6-12 Getting the text entered by the user in an editable text item

```
PROCEDURE MyHandleModelessDialogs(theEvent: EventRecord);
VAR
   myDialog:            DialogPtr;
   itemHit, itemType:   Integer;
   searchStringHandle:  Handle;
   replaceStringHandle: Handle;
   searchString:        Str255;
   replaceString:       Str255;
   itemRect:            Rect;
```

```
BEGIN
   {use DialogSelect, then determine whether the event occurred }
   { in the Global Changes dialog box; if so, respond to mouse }
   { clicks as follows}
   CASE itemHit OF
      kChange:       {user clicked the Change button}
         BEGIN
            GetDialogItem(myDialog, kFind, itemType,
                        searchStringHandle, itemRect);
            GetDialogItemText(searchStringHandle, searchString);
            GetDialogItem(myDialog, kReplace, itemType,
                        replaceStringHandle, itemRect);
            GetDialogItemText(replaceStringHandle, replaceString);
            {get a handle to the Stop button}
            GetDialogItem(myDialog, kStop, itemType,
                        itemHandle, itemRect);
            {make the Stop button active during the operation}
            HiliteControl(ControlHandle(itemHandle), 0);
            {get a handle to the Change button}
            GetDialogItem(myDialog, kChange, itemType,
                        itemHandle, itemRect);
            {make the Change button inactive during the operation}
            HiliteControl(ControlHandle(itemHandle), 255);
            DoReplace(searchString, replaceString);
            {when the operation is complete, dim Stop and make }
            { Change active here}
         END;
      kStop:      {user clicked the Stop button}
         BEGIN
                  {cancel operation, then make Stop button }
                  { inactive and Change button active again}
         END;
   END;
END;
```

In Listing 6-12, when the user clicks the Change button, the `GetDialogItem` procedure returns a handle to the item containing the search string. Because this is a handle to an editable text item, the application can pass the handle to the `GetDialogItemText` procedure, which then returns the item's text string in its second parameter. These two procedures are then used to get the string in the item containing the replacement string. These two strings are then passed to an application-defined routine that replaces all

instances of the first string with the characters of the second string. Note that when the user clicks Change, the Control Manager procedure `HiliteControl` is used to make the Stop button active and to make the Change button inactive—that is, dimmed. This indicates that the user can use the Stop button but not the Change button while the change operation is taking place.

Adding Items to an Existing Dialog Box

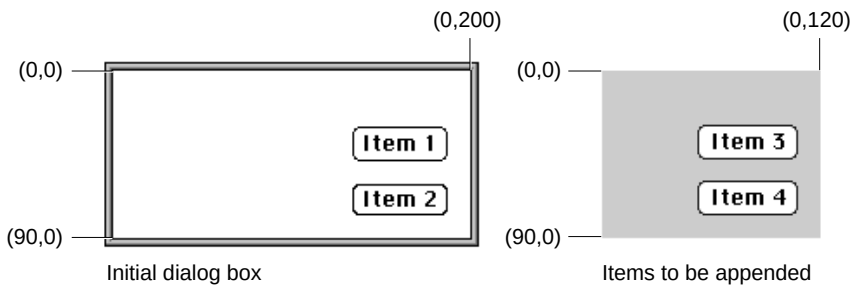
You can dynamically add items to and remove items from a dialog box by using the `AppendDITL` and `ShortenDITL` procedures. When you create a dialog box, the Dialog Manager creates a *dialog record*. The Dialog Manager then reads in the item list resource and stores a handle to it in the `items` field of the dialog record. Because every dialog box you create has its own dialog record, you can define dialog boxes whose items are defined by the same item list resource. The `AppendDITL` and `ShortenDITL` procedures are especially useful if several dialog boxes share the same item list resource and you want to add or remove items as appropriate for individual dialog boxes.

When you call the `AppendDITL` procedure, you specify a dialog box, and you specify a new item list resource to append to the dialog box's existing item list resource. You also specify where the Dialog Manager should display the new items. You can use one of these constants to designate where `AppendDITL` should display the appended items:

```
CONST overlayDITL       = 0;     {overlay existing items}
      appendDITLRight   = 1;     {append at right}
      appendDITLBottom  = 2;     {append at bottom}
TYPE DITLMethod = Integer;
```

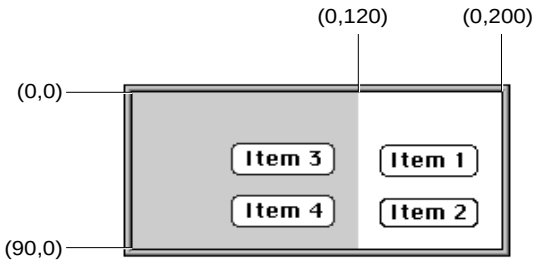
Figure 6-28 illustrates an existing dialog box and a pair of items to be appended.

Figure 6-28 An existing dialog box and items to append



If you specify the `overlayDITL` constant, `AppendDITL` superimposes the appended items over the dialog box. That is, `AppendDITL` interprets the coordinates of the display rectangles for the appended items (as specified in their item list resource) as local coordinates within the dialog box. Figure 6-29 shows the result of overlaying the items upon the dialog box illustrated in Figure 6-28.

Figure 6-29 The dialog box after items are overlaid

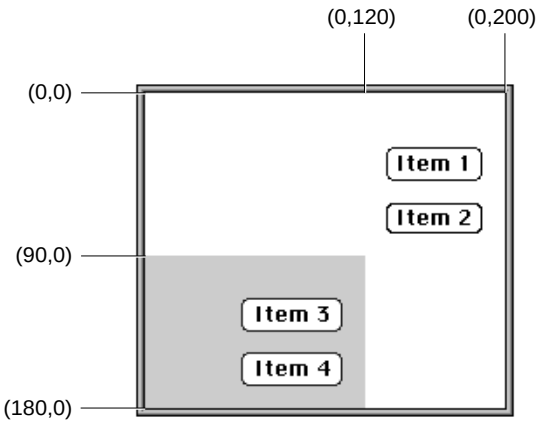


If you specify the `appendDITLRight` constant, `AppendDITL` appends the items to the right side of the dialog box, as illustrated in Figure 6-30, by positioning the display rectangles of the appended items relative to the upper-right coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.

Figure 6-30 The dialog box after items are appended to the right

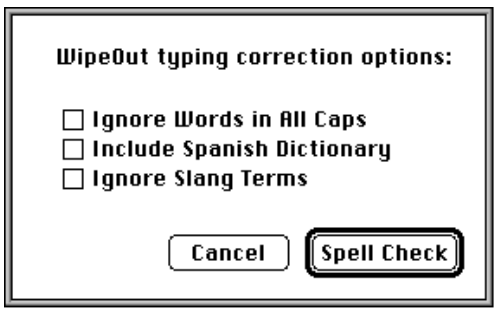
If you specify the `appendDITLBottom` constant, `AppendDITL` appends the items to the bottom of the dialog box, as illustrated in Figure 6-31, by positioning the display rectangles of the appended items relative to the lower-left coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.

Figure 6-31 The dialog box after items are appended to the bottom



As an alternative to passing the `overlayDITL`, `appendDITLRight`, or `appendDITLBottom` constant, you can pass a negative number to `AppendDITL`, which appends the items relative to an existing item in the dialog box. The absolute value of this number is interpreted as the item in the dialog box relative to which the new items are to be positioned. For example, if you pass –2 to `AppendDITL`, the display rectangles of the appended items are offset from the upper-left corner of item number 2 in the dialog box. Figure 6-12 on page 6-24 shows a simple dialog box with two checkboxes. Figure 6-32 shows the same dialog box after an additional item is appended relative to the first checkbox, so that the new item appears between the two existing checkboxes.

Figure 6-32 A dialog box with an item appended relative to an existing item



The application-defined routine called `DoSpellBoxWithSpanish`, which is shown in Listing 6-13 on the next page, illustrates the use of the `AppendDITL` procedure to add the new item.

Listing 6-13 Appending an item to an existing dialog box

```
FUNCTION DoSpellBoxWithSpanish: OSErr;
VAR
   theDialog:        DialogPtr;
   myNewItem:        Handle;
   docWindow:        WindowPtr;
   event:            EventRecord;
BEGIN
   theDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
   IF theDialog <> NIL THEN
   BEGIN
      myNewItem := GetResource('DITL', kSpanishDITL);
      IF myNewItem <> NIL THEN
      BEGIN
         AppendDITL(theDialog, myNewItem, kAppendItem);  {kAppendItem = -3}
         ReleaseResource(myNewItem);
         docWindow := FrontWindow;     {get the front window}
         {if there's a front window, deactivate it}
         IF docWindow <> NIL THEN
            DoActivate(docWindow, FALSE, event);
         ShowWindow(theDialog);  {show dialog box with appended item}
         MyAdjustMenus;             {adjust menus as needed}
         REPEAT
            ModalDialog(@MyEventFilter, itemHit);
               {handle clicks in checkboxes here}
         UNTIL ((itemHit = kSpellCheck) OR (itemHit = kCancel));
               {handle clicks in buttons here}
         DisposeDialog(theDialog);
         DoSpellBoxWithSpanish := kSuccess;
      END
      ELSE
         DoSpellBoxWithSpanish := kFailed;
   END
   ELSE DoSpellBoxWithSpanish := kFailed;
END;
```

The `DoSpellBoxWithSpanish` routine uses `GetNewDialog` to create a dialog box. As you'll see in Listing 6-14, the dialog resource passed to `GetNewDialog` has a resource ID of 402, and this dialog resource in turn specifies an item list resource with resource ID 402. The `DoSpellBoxWithSpanish` routine then uses the Resource Manager function `GetResource` to obtain a handle to a second item list resource; this item list resource contains the "Include Spanish Dictionary" checkbox. By setting a value of –3 in the last parameter of `AppendDITL`, the `DoSpellBoxWithSpanish` routine

appends the items in the second item list resource relative to item number 3 (the "Ignore Words in All Caps" checkbox) in the dialog box. Listing 6-14 shows the dialog resource for the dialog box, its regular item list resource, and the item list resource that `AppendDITL` adds to it.

Listing 6-14 Rez input for a dialog box and the item appended to it

```
# define kSpellCheckID 402   /*resource ID for Spell Check dialog box*/
# define kSpellCheckDITL 402 /*resource ID for item list resource*/
# define kSpanishDITL 257    /*resource ID for item list resource to append*/
# define kAppendHelp 257     /*resource ID for 'hdlg' for appended item*/

resource 'DLOG' (kSpellCheckID, purgeable) {/*Spell Check dialog box*/
   {62, 184, 216, 448},
   dBoxProc,               /*make it modal*/
   invisible,              /*make it initially invisible*/
   noGoAway, 0x0, kSpellCheckDITL, "Spellcheck Options",
   alertPositionParentWindow     /*place over the document window*/
};
resource 'DITL' (kSpellCheckDITL, purgeable) {
   /*items for Spell Check dialog box*/
   /*ITEM NO. 1, the "Spell Check" button, goes here*/
   /*ITEM NO. 2, the "Cancel" button, goes here*/
   /*ITEM NO. 3*/
   {48, 23, 67, 202}, CheckBox {enabled, "Ignore Words in All Caps"},
   /*ITEM NO. 4*/
   {83, 23, 101, 196}, CheckBox {enabled, "Ignore Slang Terms"},
      /*static text, help item, etc. go here*/
};
   /*add this item list resource to Spell Check dialog box only when */
   /* Spanish language dictionary is installed*/
resource 'DITL' (kSpanishDITL, purgeable) {
   {  {18, 0, 36, 209},CheckBox {enabled,"Include Spanish Dictionary"},
      {0,0,0,0}, HelpItem {disabled, HMScanAppendhdlg{kAppendHelp}} /*help*/
   }
};
```

The dialog resource specifies that the dialog box is invisible so that the application can add the new item to the dialog box before displaying it. In Listing 6-13, the `DoSpellBoxWithSpanish` routine uses the Window Manager procedure `ShowWindow` to display the dialog box after its new item has been appended. ("Displaying Alert and Dialog Boxes" beginning on page 6-61 describes more fully how to display dialog boxes.)

The appended item list resource includes a help item that causes the Help Manager to use the help resource associated with that item list resource in addition to the help resource originally associated with the dialog box. See the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for information about using the `HMScanAppendhdlg` identifier in a help item.

Listing 6-13 uses the Resource Manager procedure `ReleaseResource`. The `AppendDITL` procedure modifies the contents of the dialog box (for instance, by enlarging it). To use an unmodified version of the dialog box at a later time, your application needs to use `ReleaseResource` to release the memory occupied by the appended item list. Otherwise, if your application calls `AppendDITL` to add items to that dialog box again, the dialog box will remain modified by your previous call—for example, it will still be longer at the bottom if you previously used the `appendDITLBottom` constant.

When you can call the `ShortenDITL` procedure to remove items from the end of a dialog item list, you specify a pointer to the dialog box and the number of items to remove from the end of the item list. Note that `ShortenDITL` does not automatically resize the dialog box; you can use the Window Manager procedure `SizeWindow` if you need to resize the dialog box. You can use the `CountDITL` function to determine the number of items in the item list resource for a dialog box.

Using an Application-Defined Item to Draw the Bold Outline for a Default Button

You can define your own type of item for dialog boxes. You might wish, for example, to display a clock with the current time in a dialog box. You can also use application-defined items to draw a bold outline around the default button in a dialog box.

You should not use application-defined items in an alert box because they add unnecessary programming complications. If you need an application-defined item, use a dialog box instead.

To define your own item, include an item of type `UserItem` in your item list resource; it should have a display rectangle, but no text and no resource ID associated with it. The dialog resource that uses this item list resource must specify the `invisible` constant. This makes the dialog box invisible while you install a draw procedure for your application-defined item. After installing the procedure that draws the application-defined item, you display the dialog box by using the Window Manager procedure `ShowWindow`.

For example, Figure 6-32 on page 6-53 illustrates a dialog box that outlines the default button (Spell Check). To outline the button, the application must add an item of type `UserItem` to the item list resource for that dialog box.

So that an application-defined drawing procedure can draw a border around the Spell Check button, the item list resource in Listing 6-15 specifies a larger display rectangle for the application-defined item than for the Spell Check button.

Listing 6-15 Rez input for an application-defined item in an item list

```
resource 'DITL' (kSpellCheckDITL, purgeable) {
   /*ITEM NO. 1: OK button--the default*/
   {  {123, 170, 144, 254}, Button {enabled,"Spell Check"},
   /*ITEMs 2-5 go here: Cancel button, two checkboxes, and static text*/
   /*ITEM NO. 6: application-defined item*/
      {115, 164, 152, 260},   /*6th item*/
      UserItem {              /*draw procedure for item draws an outline*/
            disabled,         /*1st item lies inside this--1st is enabled*/
      }
   }
};
```

The application-defined item is disabled because the Spell Check button, which lies within the application-defined item, is enabled. Because the Spell Check button is listed before the application-defined item in this item list resource, the Dialog Manager reports when the user clicks the Spell Check button. However, note that when application-defined items are enabled, the Dialog Manager reports their item numbers when the user clicks them.

Note
Although the draw procedure for an application-defined item can draw outside the item's display rectangle, this is not recommended because if the Dialog Manager receives an update event involving an area outside the display rectangle but inside the area where you draw your application-defined item, the Dialog Manager won't call your draw procedure. ◆

Listing 6-14 on page 6-55 shows the dialog resource for the dialog box. Notice that the `invisible` constant in the dialog resource specifies that the dialog box should initially be invisible.

You must provide a procedure that draws your application-defined item. Your draw procedure must have two parameters: a dialog pointer and an item number from the dialog box's item list resource. For example, this is how you should declare the draw procedure if you were to name it `MyDrawDefaultButtonOutline`:

```
PROCEDURE MyDrawDefaultButtonOutline (theDialog: DialogPtr;
                                      theItem: Integer);
```

The parameter `theDialog` is a pointer to the dialog box containing the application-defined item. (If your procedure draws in more than one dialog box, this parameter tells your procedure which dialog box to draw in.) The parameter `theItem` is a number corresponding to the position of an item in the item list resource for the dialog box. (In case the procedure draws more than one item, this parameter tells the procedure which one to draw.)

To install this draw procedure, use the `GetDialogItem` and `SetDialogItem` procedures. Use `GetDialogItem` to return a handle to the application-defined item specified in the item list resource. Then use `SetDialogItem` to replace this handle with a pointer to your draw procedure. When calling your draw procedure, the Dialog Manager sets the current port to the dialog box's graphics port. The Dialog Manager then calls your procedure to draw the application-defined item as necessary—for instance, when you display the dialog box and whenever the Dialog Manager receives an update event for the dialog box.

Listing 6-16 illustrates how to install the procedure that draws a bold outline. In this listing, `GetDialogItem` gets a handle to the application-defined item (which is the sixth item in the item list resource from Listing 6-15). The procedure pointer `@MyDrawDefaultButtonOutline`, which is coerced to a handle, is then passed to `SetDialogItem`, which sets the draw procedure into the dialog record.

Listing 6-16 Installing the draw procedure for an application-defined item

```
FUNCTION DisplayMyDialog (VAR theDialog: DialogPtr): OSErr;
VAR
   itemType:      Integer;
   itemHandle:    Handle;
   itemRect:      Rect;
   docWindow:     WindowPtr;
   event:      EventRecord;
BEGIN
   {begin by creating an invisible dialog box}
   theDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
   IF theDialog <> NIL THEN
   BEGIN
      {get a handle to the application-defined item (i.e., userItem)}
      GetDialogItem(theDialog, kUserItem, itemType, itemHandle, itemRect);
      {install the drawing procedure for the application-defined item}
      SetDialogItem(theDialog, kUserItem, itemType,
                    Handle(@MyDrawDefaultButtonOutline), itemRect);
      docWindow := FrontWindow;     {get the front window}
         {if there's a front window, deactivate it}
      IF docWindow <> NIL THEN
         DoActivate(docWindow, FALSE, event);
      ShowWindow(theDialog);     {display the dialog box}
      MyAdjustMenus;             {adjust menus as needed}
      DisplayMyDialog := kSuccess;
   END
   ELSE DisplayMyDialog := kFailed;
   {call ModalDialog and handle events in dialog box here}
END;
```

Use the Window Manager procedure `ShowWindow` to display the previously invisible dialog box. When `ShowWindow` is called in this example, a bold outline is drawn inside the application-defined item and around the Spell Check button.

Listing 6-17 shows a procedure that draws a bold outline around a button of any size and shape. This procedure can be used to draw the outline around the Spell Check button from the previous example.

Listing 6-17 Creating a draw procedure that draws a bold outline around the default button

```
PROCEDURE MyDrawDefaultButtonOutline(theDialog: DialogPtr; theItem: Integer);
CONST
   kButtonFrameInset =  -4;
   kButtonFrameSize =   3;
   kCntrActivate =      0;
VAR
   itemType:      Integer;    {returned item type}
   itemRect:      Rect;       {returned display rectangle}
   itemHandle:    Handle;     {returned item handle}
   curPen:        PenState;
   buttonOval:    Integer;
   fgSaveColor:   RGBColor;
   bgColor:       RGBColor;
   newfgColor:    RGBColor;
   newGray:       Boolean;
   oldPort:       WindowPtr;
   isColor:       Boolean;
   targetDevice:  GDHandle;
BEGIN
   {get the default button & draw a bold border around it}
   GetDialogItem(theDialog, kDefaultButton, itemType, itemHandle, itemRect);
   GetPort(oldPort);
   SetPort(ControlHandle(itemHandle)^^.contrlOwner);
   GetPenState(curPen);
   PenNormal;
   InsetRect(itemRect, kButtonFrameInset, kButtonFrameInset);
   FrameRoundRect(itemRect, 16, 16);
   buttonOval := (itemRect.bottom - itemRect.top) DIV 2 + 2;
   IF ((CGrafPtr(ControlHandle(itemHandle)^^.contrlOwner)^.portVersion) =
       kIsColorPort) THEN
      isColor := TRUE
   ELSE
      isColor := FALSE;
   IF (ControlHandle(itemHandle)^^.contrlHilite <> kCntrlActivate) THEN
```

```
   BEGIN    {control is dimmed, so draw gray default button outline}
      newGray := FALSE;
      IF isColor THEN
      BEGIN
         GetBackColor(bgColor);
         GetForeColor(fgSaveColor);
         newfgColor := fgSaveColor;
            {get the device on which this dialog box is displayed}
         targetDevice :=
               MyGetDeviceFromRect(ControlHandle(itemHandle)^^.contrlRect);
            {use the gray defined by the display device}
         newGray := GetGray(targetDevice, bgColor, newfgColor);
      END;
      IF newGray THEN
         RGBForeColor(newfgColor)
      ELSE
         PenPat(gray);
      PenSize(kButtonFrameSize, kButtonFrameSize);
      FrameRoundRect(itemRect, buttonOval, buttonOval);
      IF isColor THEN
         RGBForeColor(fgSaveColor);
   END
   ELSE  {control is active, so draw default button outline in black}
   BEGIN
      PenPat(black);
      PenSize(kButtonFrameSize, kButtonFrameSize);
      FrameRoundRect(itemRect, buttonOval, buttonOval);
   END;
   SetPenState(curPen);
   SetPort(oldPort);
END;
```

Listing 6-17 uses `GetDialogItem` to get the Spell Check button and then uses several QuickDraw routines to draw a black outline around that button's display rectangle when the button is active. If the button is inactive (that is, dimmed), `MyDrawDefaultButtonOutline` draws a gray outline.

Before drawing a gray outline, `MyDrawDefaultButtonOutline` determines whether the dialog box uses a color graphics port. As explained in "Including Color in Your Alert and Dialog Boxes" beginning on page 6-75, you can supply a dialog box with a color graphics port by creating a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource. If the dialog box uses a color graphics port, `MyDrawDefaultButtonOutline` uses the Color QuickDraw function `GetGray` to return a blended gray based on the foreground and background colors. Then