CHAPTER 6

Dialog Manager

Contents

Introduction to Alerts and Dialog Boxes 6-6
- Types of Alerts 6-8
- Types of Dialog Boxes 6-9
  - Modal Dialog Boxes 6-10
  - Movable Modal Dialog Boxes 6-11
  - Modeless Dialog Boxes 6-12
- Items in Alert and Dialog Boxes 6-13
- Events in Alert and Dialog Boxes 6-14
- Alert Boxes, Dialog Boxes, and the Window Manager 6-15
- About the Dialog Manager 6-16

Using the Dialog Manager 6-17
- Creating Alert Sounds and Alert Boxes 6-18
- Creating Dialog Boxes 6-23
- Providing Items for Alert and Dialog Boxes 6-26
  - Item Types 6-30
  - Display Rectangles 6-32
  - Enabled and Disabled Items 6-36
  - Resource IDs for Items 6-36
  - Titles for Buttons, Checkboxes, and Radio Buttons 6-37
  - Text Strings for Static Text and Editable Text Items 6-40
  - Pop-Up Menus as Items 6-42
  - Keyboard Navigation Among Items 6-44
- Manipulating Items 6-44
  - Changing Static Text 6-46
  - Getting Text From Editable Text Items 6-48
  - Adding Items to an Existing Dialog Box 6-51
- Using an Application-Defined Item to Draw the Bold Outline for a Default Button 6-56

Displaying Alert and Dialog Boxes 6-61
Positioning Alert and Dialog Boxes 6-62
Deactivating Windows Behind Alert and Modal Dialog Boxes 6-64
Displaying Modeless Dialog Boxes 6-66
Adjusting Menus for Modal Dialog Boxes 6-68
Adjusting Menus for Movable Modal and Modeless Dialog Boxes 6-73
Displaying Multiple Alert and Dialog Boxes 6-74
Displaying Alert and Dialog Boxes From the Background 6-74
Including Color in Your Alert and Dialog Boxes 6-75
Handling Events in Alert and Dialog Boxes 6-77
Responding to Events in Controls 6-78
Responding to Events in Editable Text Items 6-79
Responding to Events in Alert Boxes 6-81
Responding to Events in Modal Dialog Boxes 6-82
Writing an Event Filter Function for Alert and Modal Dialog Boxes 6-86
Responding to Mouse Events in Modeless and Movable Modal Dialog Boxes 6-89
Responding to Keyboard Events in Modeless and Movable Modal Dialog Boxes 6-94
Responding to Activate and Update Events in Modeless and Movable Modal Dialog Boxes 6-97
Closing Dialog Boxes 6-100
Dialog Manager Reference 6-101
Data Structure 6-101
The Dialog Record 6-101
Dialog Manager Routines 6-102
Initializing the Dialog Manager 6-102
Creating Alerts 6-105
Creating and Disposing of Dialog Boxes 6-113
Manipulating Items in Alert and Dialog Boxes 6-120
Handling Text in Alert and Dialog Boxes 6-129
Handling Events in Dialog Boxes 6-135
Application-Defined Routines 6-143
Resources 6-147
The Dialog Resource 6-148
The Alert Resource 6-150
The Item List Resource 6-151
The Dialog Color Table Resource 6-156
The Alert Color Table Resource 6-157
The Item Color Table Resource 6-158
Summary of the Dialog Manager 6-165
Pascal Summary 6-165
Constants 6-165
Data Types 6-166
Dialog Manager Routines 6-166
Application-Defined Routines 6-168

C Summary 6-168
Constants 6-168
Data Types 6-169
Dialog Manager Routines 6-170
Application-Defined Routines 6-172
Assembly-Language Summary 6-172
Data Structures 6-172
Global Variables 6-172

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.

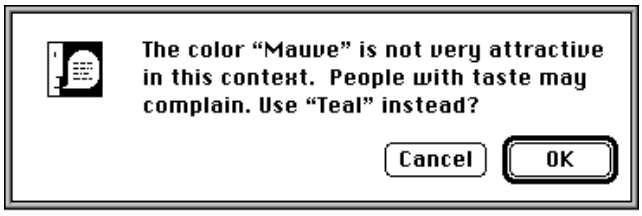
Use an alert sound for errors that are both minor and immediately obvious. For example, if the user tries to backspace past the left boundary of a text field, your application might play the alert sound instead of displaying an alert box. Your application can base its response on the number of consecutive 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.

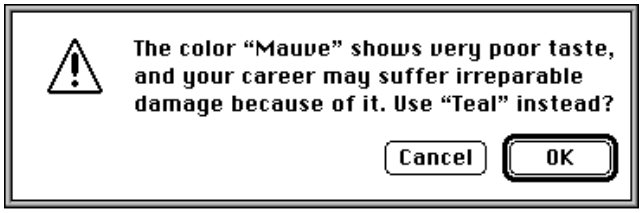
Use a **note alert** to inform users of a situation that won't have any disastrous consequences if left as is. Usually this type of alert simply offers information, and the user responds by clicking the OK button. Occasionally, as shown in Figure 6-3, a note alert may ask a simple question and provide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

Figure 6-3 A note alert



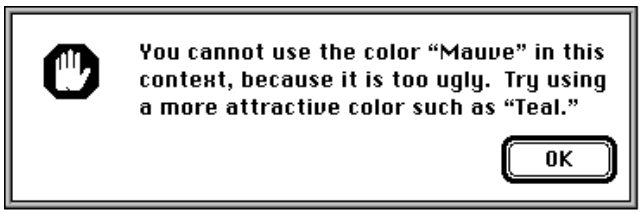
Use a **caution alert** to alert the user to an operation that may have undesirable results if it's allowed to continue. As shown in Figure 6-4, you should give the user the choice of whether to continue the action (by clicking the OK button) or to stop the action (by clicking the Cancel button).

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.

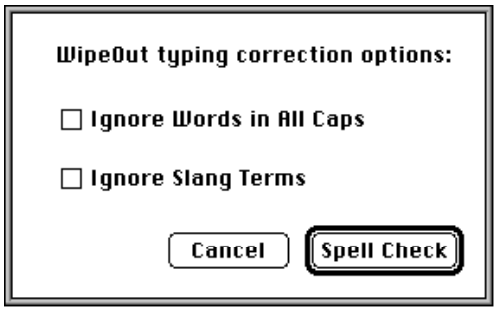
Types of Dialog Boxes

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.

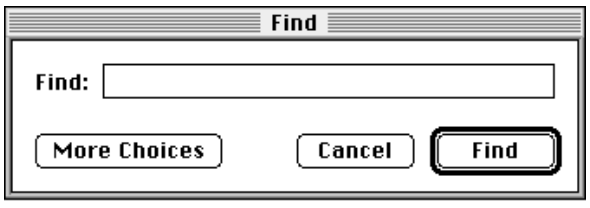
Movable Modal Dialog Boxes

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



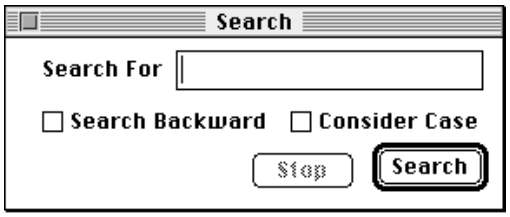
It's important to consider whether you can use a modeless dialog box instead of a modal or a movable modal dialog box—especially to preserve the user's ability to perform any task in any order.

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



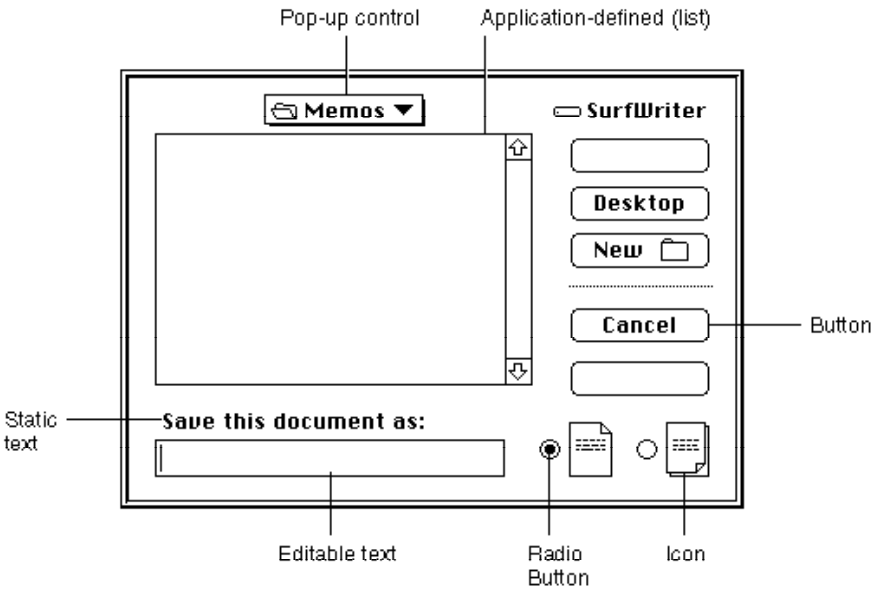
When you display a modeless dialog box, you must allow the user to perform other operations—such as working in document windows—without dismissing the dialog box. When a user clicks a button in a modeless dialog box, your application should *not* remove it; instead, the dialog box should remain on the 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

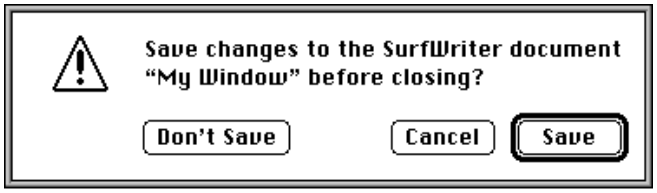
To create an alert, use one of these functions: `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert`. Icons associated with the first three functions appear in the upper-left corner of the alert boxes, as previously shown in Figure 6-3, Figure 6-4, and Figure 6-5. The `Alert` function allows you to 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.

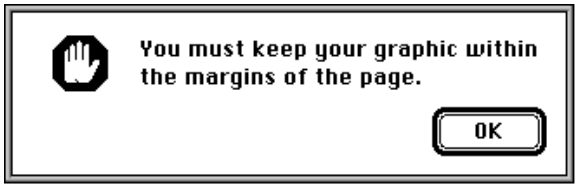
In Listing 6-1, the application-defined constant `kSaveAlertDITL` represents the resource ID for the item list resource. "Providing Items for Alert and Dialog Boxes" beginning on page 6-26 describes how to create an item list resource.

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.

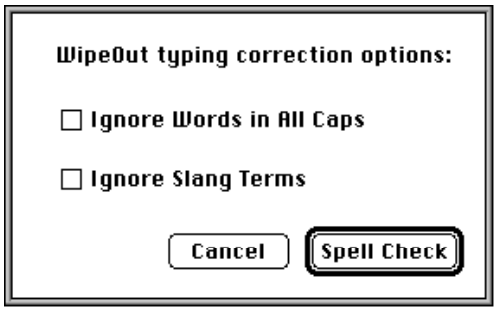
When you use `GetNewDialog`, you pass it the resource ID of the dialog resource, an optional pointer to the memory to use for the dialog record, and the window pointer `Pointer(-1)`, which causes the Window Manager to display the dialog box in front of all other windows.

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.)

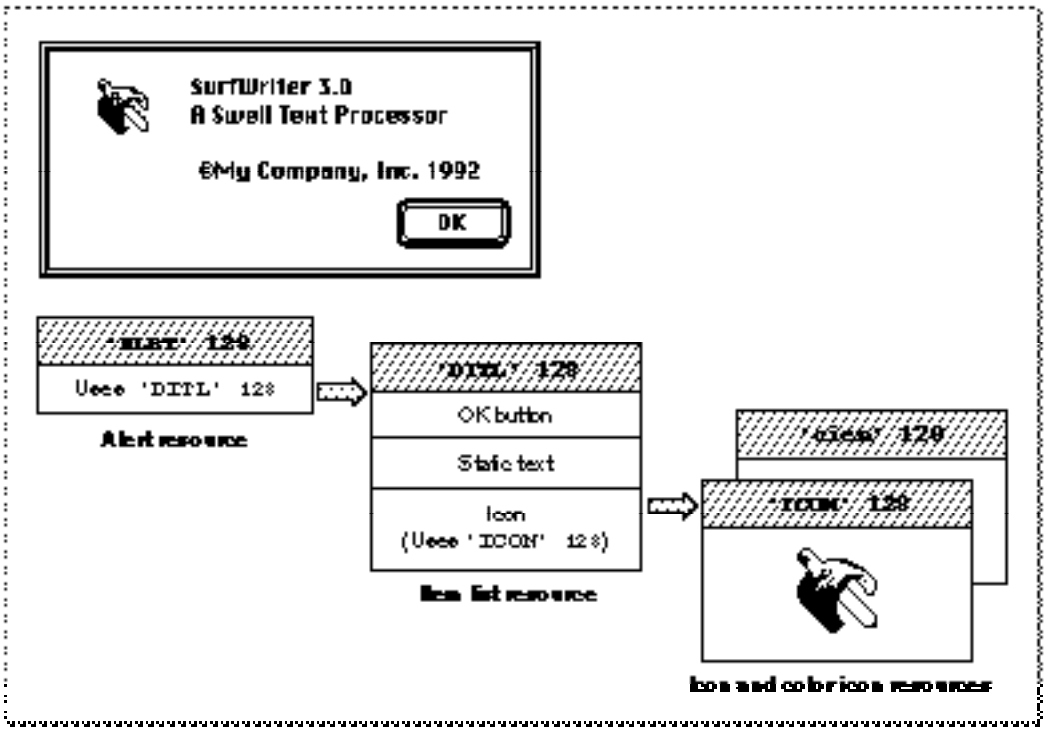
In this example, when the user chooses the About command, the SurfWriter application uses the `Alert` function to display the alert.

```
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;
```

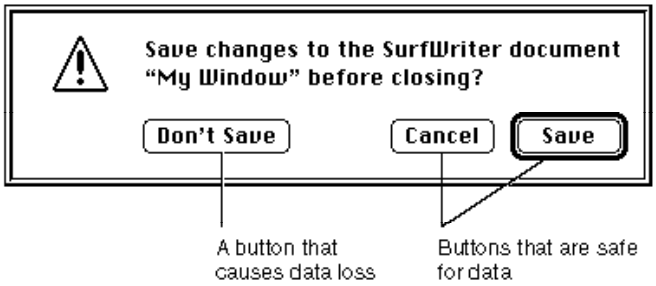
As described in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18, you define within the alert resource whether the OK or the Cancel button is the default button for each alert stage. The Dialog Manager automatically draws a bold outline around the button that you specify and, if the user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

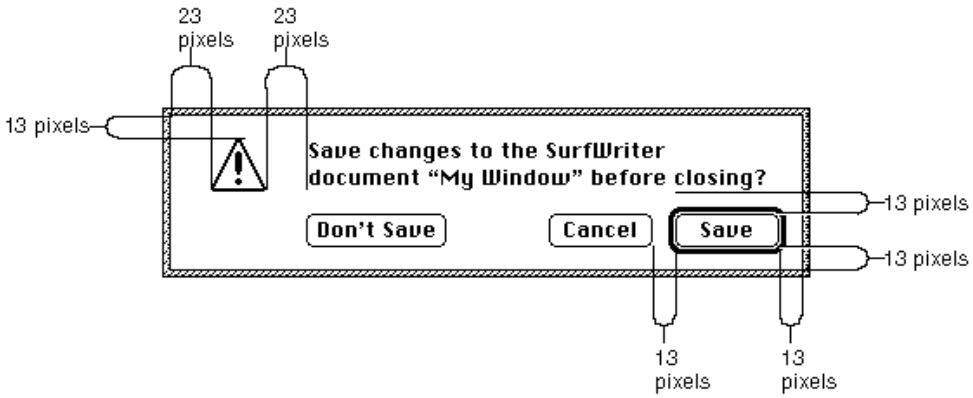
For an icon or a QuickDraw picture larger than its display rectangle, the Dialog Manager scales the icon or picture to fit the display rectangle.

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.

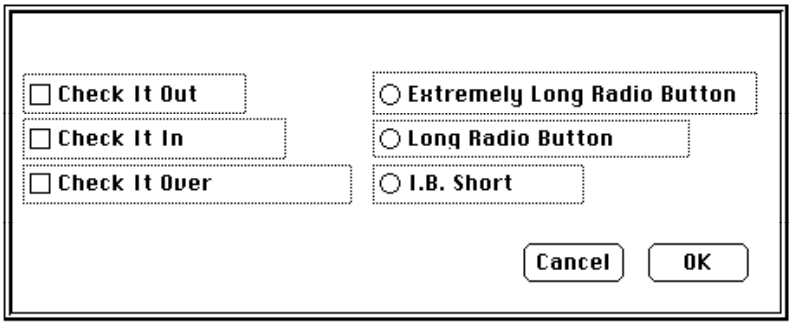
The alignment of the items in an alert box or a dialog box may vary with localization. Although in Roman script systems these items are generally aligned left to right, items in Arabic or Hebrew script systems should generally be aligned right to left, because Arabic and Hebrew are written from right 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.

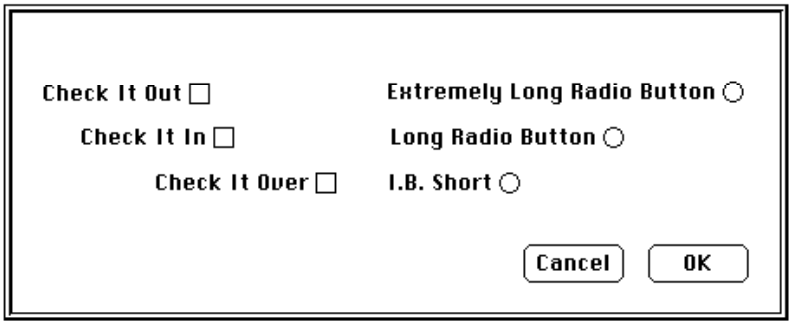
The dialog box at the top of Figure 6-16 shows several checkboxes and radio buttons with display rectangles of different sizes. The next dialog box in the figure illustrates what happens to the alignment of these items after the Control Manager transposes the controls with their titles.

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.

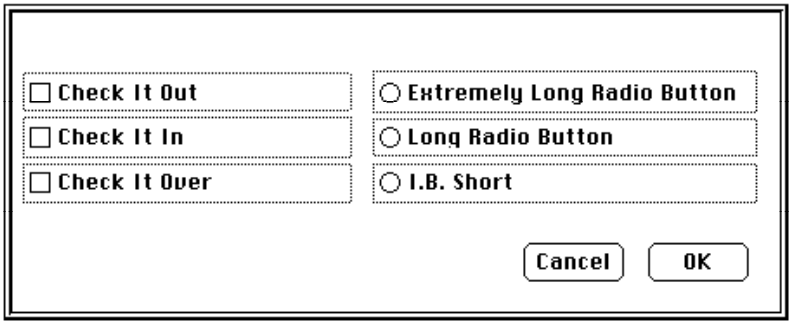
Figure 6-16 Incorrectly and correctly sized display rectangles for alternate script systems



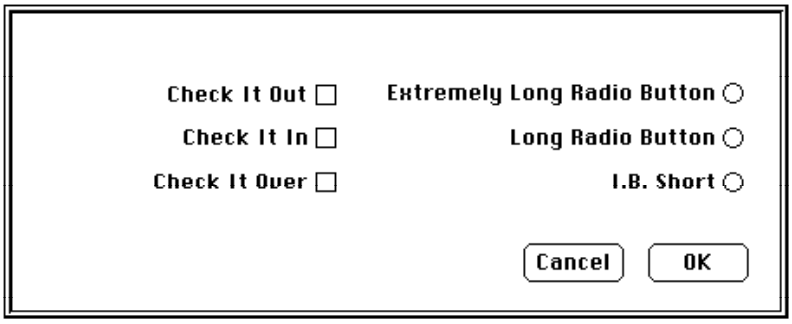
A dialog box containing display rectangles of different sizes



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*.

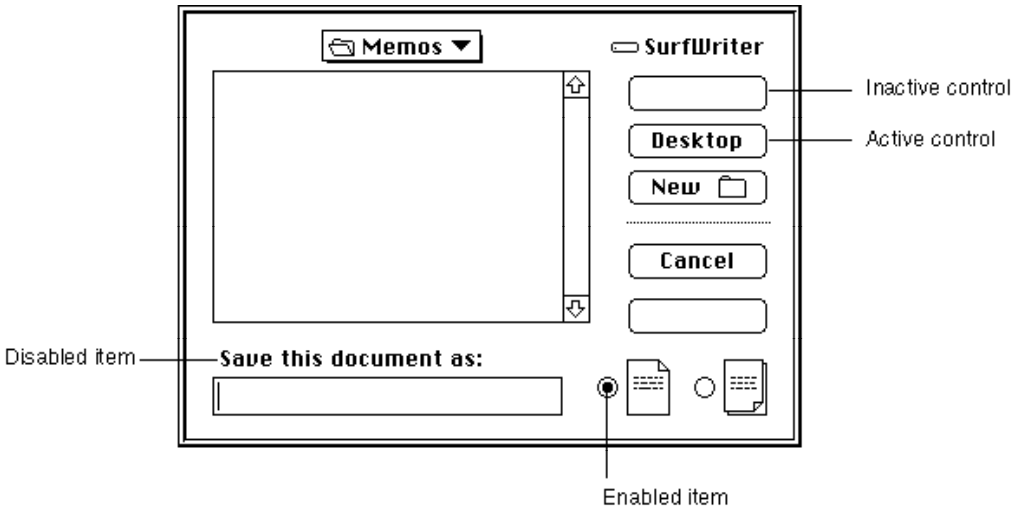
The Control Manager displays an inactive control in a way (dimmed, for example) that shows it's inactive, whereas the Dialog Manager makes no visual distinction between a disabled item and an enabled item. Figure 6-17 shows the difference between an inactive and an active control. The Control Manager 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.

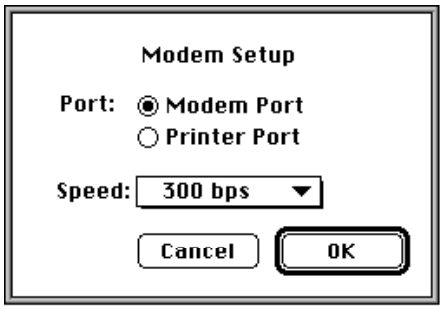
Use book-title capitalization for these items. In general, this means that you capitalize one-word titles and, in multiple-word titles, words of four or more letters. Usually you don't capitalize words such as *in*, *an*, or *and.* The rules for capitalization of titles appear in the *Apple Publications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

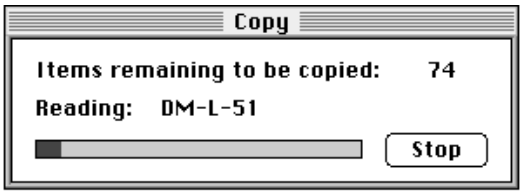
Your application should not remove a modeless dialog box when the user clicks a button; rather, you should remove the dialog box when the user clicks its close box or chooses Close from the File menu while the modeless dialog box is active.

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

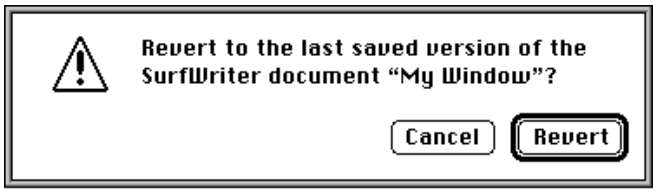
button; clicking the button halts the current file copy operation but leaves intact the copies that were previously made.

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

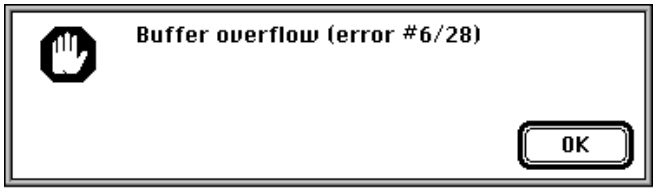


Figure 6-22 shows a somewhat better alert message. Although the vocabulary is less technical, no remedy to the problem is offered.

Figure 6-22 A less obscure alert message

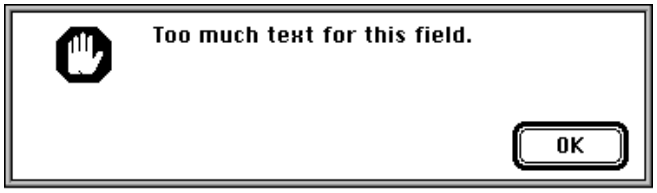


Figure 6-23 illustrates a good alert message. The message is specific, it's expressed in nontechnical terms, it explains why the error occurred, and it suggests a solution to the problem.

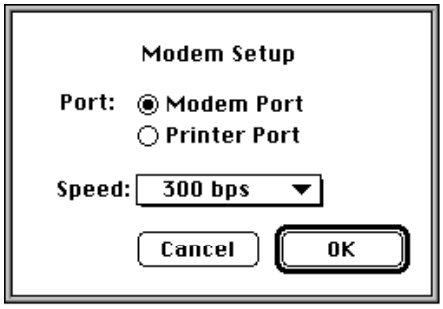
Figure 6-23 A clear and helpful alert message

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.

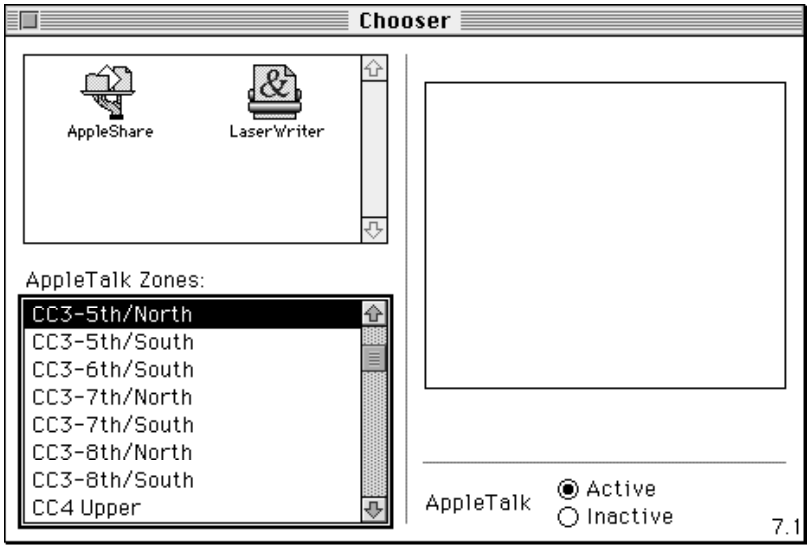
Because all typing goes to the active window, there should be only one active area and only one indicator at any time. If only one element in a dialog box can accept keyboard input and that element is a scrolling list, it's not necessary to place a border around it.

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.

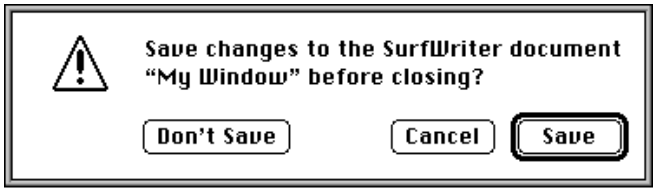
Routine	Description
`SelectDialogItemText`	Selects the text of an editable text item.
`SetDialogItem`	Sets the item type and the display rectangle of an item, or (for application-defined items) the draw procedure of an item.
`ShortenDITL`	Removes items from a dialog box.
`ShowDialogItem`	Redisplays the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

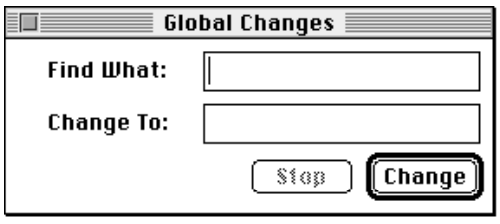
Listing 6-10 Specifying where `ParamText` should substitute text in an alert box message

```
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

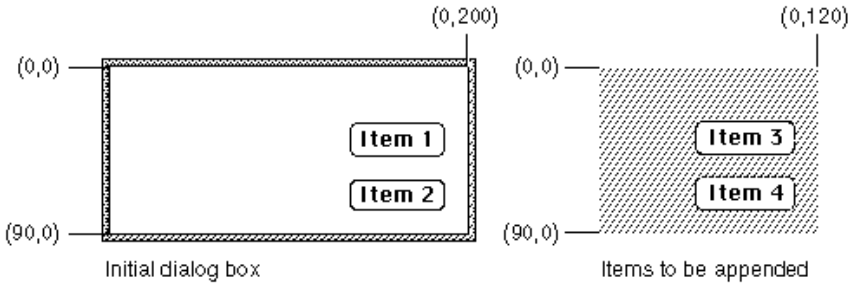
You can dynamically add items to and remove items from a dialog box by using the `AppendDITL` and `ShortenDITL` procedures. When you create a dialog box, the Dialog Manager creates a *dialog record*. The Dialog Manager then reads in the item list resource and stores a handle to it in the `items` field 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;
```

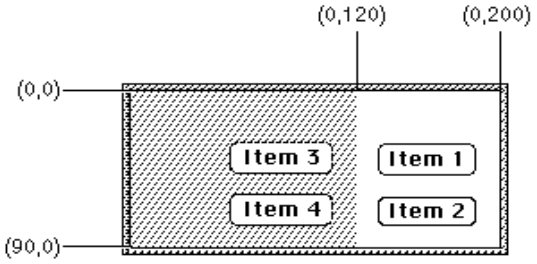
Figure 6-28 illustrates an existing dialog box and a pair of items to be appended.

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



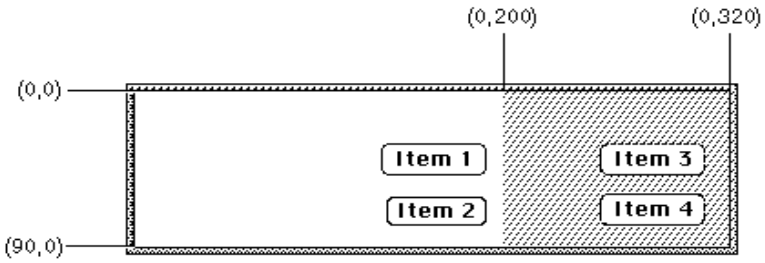
If you specify the `overlayDITL` constant, `AppendDITL` superimposes the appended items over the dialog box. That is, `AppendDITL` interprets the coordinates of the display rectangles for the appended items (as specified in their item list resource) as local coordinates within the dialog box. Figure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



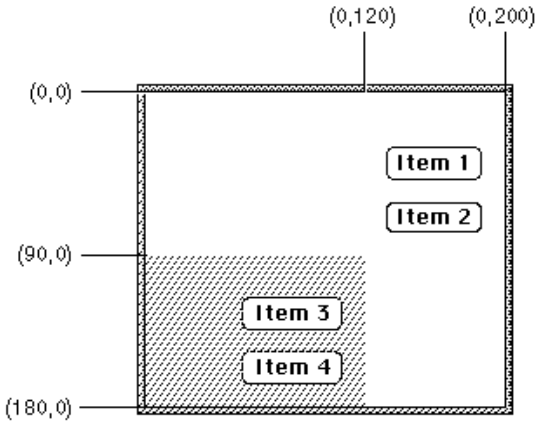
If you specify the `appendDITLRight` constant, `AppendDITL` appends the items to the right side of the dialog box, as illustrated in Figure 6-30, by positioning the display rectangles of the appended items relative to the upper-right coordinate of the dialog box. The `AppendDITL` procedure automatically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

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



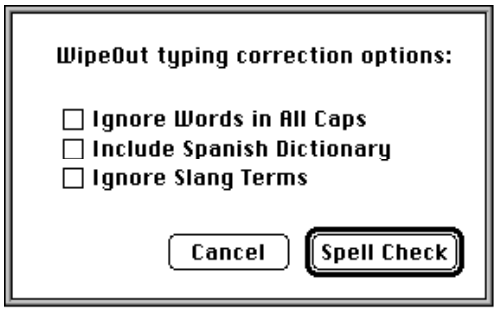
If you specify the `appendDITLBottom` constant, `AppendDITL` appends the items to the bottom of the dialog box, as illustrated in Figure 6-31, by positioning the display rectangles of the appended items relative to the lower-left coordinate of the dialog box. The `AppendDITL` procedure automatically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

`MyDrawDefaultButtonOutline` uses this gray for outlining the dimmed default button. Otherwise, `MyDrawDefaultButtonOutline` uses the QuickDraw procedure `PenPat` to draw a gray outline on black-and-white monitors.

Displaying Alert and Dialog Boxes

You typically define alerts and dialog boxes in resources, as described in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 and in "Creating Dialog Boxes" beginning on page 6-23. To create an alert or a dialog box, you use a Dialog Manager function—such as `Alert` or `GetNewDialog`—that incorporates information from your item list resource and from your alert resource or dialog resource into a data structure, called a *dialog record*, in memory. The Dialog Manager creates a dialog record, which is a data structure of type `DialogRecord`, whenever your application creates an alert or a dialog box.

The Dialog Manager automatically displays alert boxes at the appropriate alert stages; it also automatically displays those dialog boxes that you specify as visible in their dialog resources. But you must use a Window Manager routine such as `ShowWindow` to display dialog boxes that you specify as invisible in their dialog resources.

When you use a function that creates an alert (namely, `Alert`, `StopAlert`, `NoteAlert`, or `CautionAlert`), the Dialog Manager automatically displays the alert box at the alert stages that you specify with the `visible` constant in your alert resource. You do not use any routines other than the `Alert`, `StopAlert`, `NoteAlert`, and `CautionAlert` functions to display an alert box.

When you specify the `visible` constant in a dialog resource, the Dialog Manager immediately displays the dialog box when you use the `GetNewDialog` function. If you instead specify the `invisible` constant so that the dialog box is initially invisible when you call `GetNewDialog`, use the Window Manager procedure `ShowWindow` to display it. This is useful if you need to manipulate a dialog item dynamically using `GetDialogItem` and `SetDialogItem` before you display the dialog box. For example, if you want to install an application-defined draw procedure for a dialog box, you specify the `invisible` constant in a dialog resource, pass the resource ID of that dialog resource in a parameter to `GetNewDialog`, use `GetDialogItem` and `SetDialogItem` to install the application-defined draw procedure, then call `ShowWindow` to display the dialog box, as previously shown in Listing 6-16 on page 6-58.

You should always specify `Pointer(-1)` as a parameter to `GetNewDialog` to display a dialog box as the active (that is, frontmost) window.

You should perform the following tasks in conjunction with displaying an alert box or a dialog box:

- Specify an appropriate screen position at which to display the alert box or dialog box.
- Deactivate the frontmost window (if one exists) before displaying an alert box or a modal dialog box.
- Determine whether you've already created a modeless dialog box and, if so, select it instead of creating a new instance of it.
- Adjust your menus appropriately for a modal dialog box with editable text items and for any movable modal and modeless dialog box you wish to display.

The `DialogSelect` function uses the QuickDraw procedure `SetPort` to make the alert or dialog box the current graphics port. The `ModalDialog` procedure and the functions that create alert boxes use `DialogSelect` to respond to update and activate events. You can also use `DialogSelect` to respond to update and activate events in your modeless and movable modal dialog boxes. In response to update events, you can instead use the `UpdateDialog` function, which also makes the dialog box the current graphics port. In these cases, it's generally not necessary for your application to call `SetPort` when displaying, updating, or activating alert boxes and dialog boxes. See *Inside Macintosh: Imaging* for more information about `SetPort`.

These and other related issues are explained in detail in the next several sections of this chapter.

Positioning Alert and Dialog Boxes

As previously described in "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 and "Creating Dialog Boxes" beginning on page 6-23, you specify a rectangle in every alert resource and dialog resource. The dimensions of this rectangle determine the dimensions of the alert box or dialog box. You can also let the rectangle coordinates serve as the global coordinates that determine the position of the alert box or dialog box, or you can let the Dialog Manager automatically locate it for you according to three standard positions. To specify these standard positions in System 7, your application can use the following constants in the Rez input files for alert resources and dialog resources:

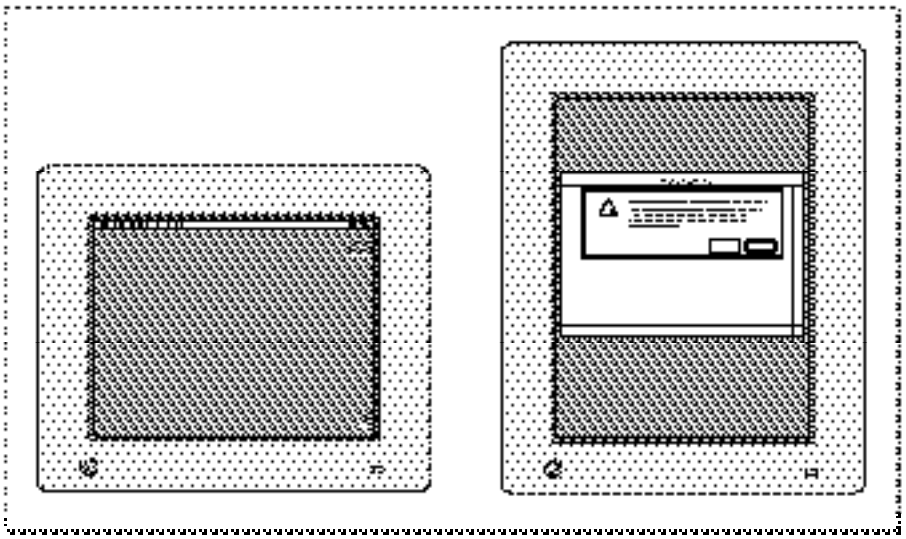
Constant	Description
`alertPositionParentWindow`	Position the alert or dialog box over the frontmost window
`alertPositionMainScreen`	Position the alert or dialog box on the main screen
`alertPositionParentWindowScreen`	Position the alert or dialog box on the screen containing the frontmost window

If your application positions alert or dialog boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager. If you do use these constants, use them to specify the positions of both alert boxes and dialog boxes.

The next three figures illustrate various alert boxes that might appear when the user is working on two monitors: a 12-inch monitor (the main screen) that displays the menu bar and a full-page monitor that displays a document window. These figures show where the Dialog Manager places an alert box according to the position specified in the alert resource.

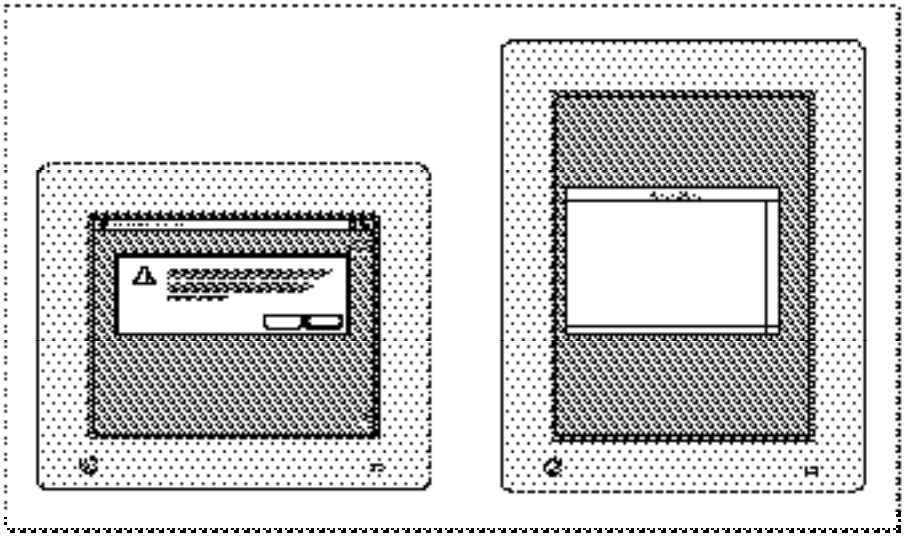
Figure 6-33 shows an alert box displayed in response to an error made by the user while working on a document; the alert resource specifies the `alertPositionParentWindow` constant, which tells the Dialog Manager to position the alert box over the frontmost window so that the window's title bar appears. This position is appropriate for an alert box or a dialog box that relates directly to the frontmost window. You should always try to position alert boxes and dialog boxes where the user is working.

Figure 6-33 An alert box in front of a document window



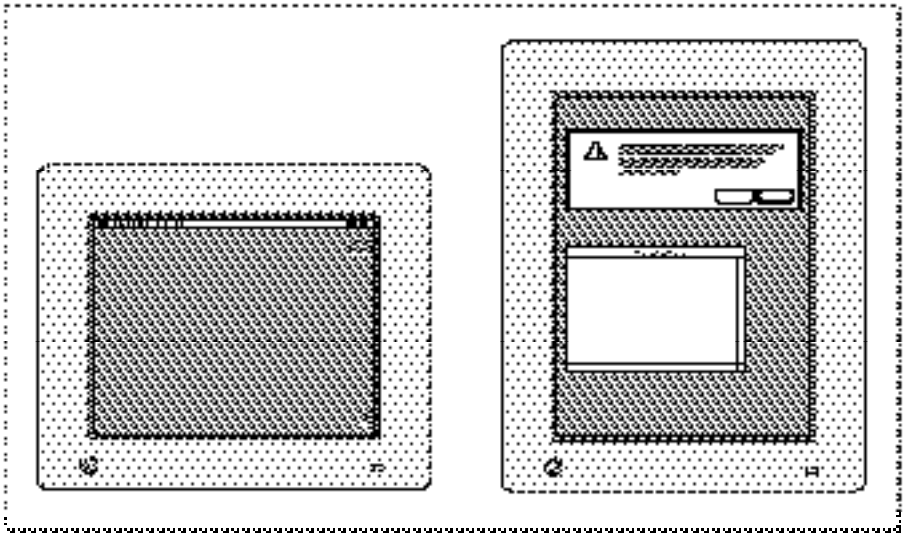
Not all alert boxes or dialog boxes relate to the frontmost window. Some may relate only to actions the user performs on the main screen. For example, Figure 6-34 illustrates an alert box displayed when the user chooses the About command from the Apple menu. For an alert box or dialog box such as this, you should specify the `alertPositionMainScreen` constant in the alert or dialog resource. Figure 6-34 shows how the Dialog Manager centers such an alert box near the top of the main screen.

Figure 6-34 An alert box on the main screen



Sometimes you may need to display an alert box or a dialog box that applies neither to the frontmost window nor to an action performed on the main screen. To catch the user's attention, you should position such an alert or dialog box on the screen where the user is working. For example, if you need to alert the user that available disk space is low, you should specify the `alertPositionParentWindowScreen` constant. Figure 6-35 shows how the Dialog Manager displays such an alert box or dialog box when a document window appears on a screen other than the main screen.

Figure 6-35 An alert box in the alert position of the document window screen



If you don't specify a positioning constant, the Dialog Manager uses the rectangle coordinates in your alert resource or dialog resource as global coordinates specifying where to position your alert or dialog box. If you wish to specify the position yourself in this manner, you should generally try to center alert and dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is most appropriate. If you don't use the positioning constants, you should also place the tops of alert and dialog boxes (including the title bars of modeless and movable modal dialog boxes) below the menu bar. You can use the `GetMBarHeight` function, described in the chapter "Menu Manager" in this book, to determine the height of the menu bar.

Deactivating Windows Behind Alert and Modal Dialog Boxes

For alert and modal dialog boxes, the `ModalDialog` procedure traps all events before they are passed to your event loop, which normally handles activate events for your windows. Thus, if a window is active, you must explicitly deactivate it before displaying an alert box or a modal dialog box.

Your modeless dialog boxes and movable modal dialog boxes never use the `ModalDialog` procedure. Therefore, you do not have to deactivate the frontmost window explicitly before displaying a modeless or a movable modal dialog box.

Instead, the Event Manager continues sending your application activate events for your windows as needed, which you typically handle in your normal event loop. (The chapters "Event Manager" and "Window Manager" in this book explain how to activate and deactivate windows.)

Plate 2 at the front of this book shows an alert box that an application displays when the user chooses the About command in the Apple menu. Listing 6-18 shows an application-defined routine, `ShowMyAboutBox`, that displays this alert box.

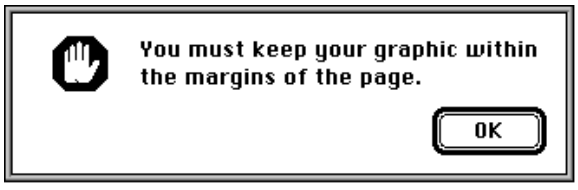
Listing 6-18 Deactivating the front window before displaying an alert box

```
PROCEDURE ShowMyAboutBox;
VAR
   itemHit:    Integer;
   docWindow:  WindowPtr;
   event:      EventRecord;
BEGIN
   docWindow := FrontWindow;     {get the front window}
   {if there's a front window, deactivate it}
   IF docWindow <> NIL THEN
      DoActivate(docWindow, FALSE, event);
   {then show the alert box}
   itemHit := Alert(kAboutBoxID, @MyEventFilter);
END;
```

The `ShowMyAboutBox` routine uses the Window Manager function `FrontWindow`. If `FrontWindow` returns a valid pointer, `ShowMyAboutBox` calls its `DoActivate` procedure to deactivate that window before calling the `Alert` function to display the alert box. When the user clicks the OK button, the alert box is dismissed. The Event Manager then sends the application update events so that it can update the contents of any windows as appropriate, and the Event Manager sends the application an activate event so that it can activate the previously frontmost window again. The application handles these events in its normal event loop.

If your application does not display an alert box during certain alert stages, use the `GetAlertStage` function to test for those stages before deactivating the active window. The `GetAlertStage` function returns the last occurrence of an alert as a number from 0 to 3. Figure 6-36 shows an alert box that appears only after the user repeats an error three consecutive times.

Figure 6-36 An alert box displayed only after the third alert stage



Listing 6-19 shows how you might use `GetAlertStage` to determine if such an alert needs to be displayed before deactivating the document window.

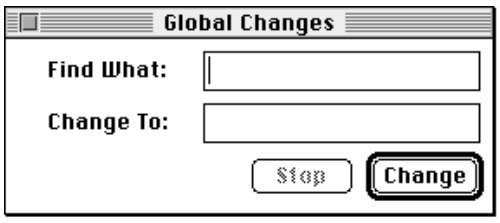
Listing 6-19 Using `GetAlertStage` to determine when to deactivate the front window

```
PROCEDURE MyAlert;
VAR
   itemHit:       Integer;
   alertStage:    Integer;
   docWindow:     WindowPtr;
   event:         EventRecord;
BEGIN
   docWindow := FrontWindow;
   alertStage := GetAlertStage;
   IF (alertStage >= 2) AND (docWindow <> NIL) THEN    {at 3rd alert stage, }
      DoActivate(docWindow, FALSE, event);      { deactivate front window & }
   itemHit := StopAlert(kStopAlertID, @MyEventFilter);   { display alert box}
END;
```

Displaying Modeless Dialog Boxes

For a modeless dialog box, check to make sure it isn't already open before you create and display it. For example, the modeless dialog box shown in Figure 6-37 should appear when the user chooses the Global Changes command. After invoking this command, the user may select another window, thereby deactivating the modeless dialog box.

Figure 6-37 A modeless dialog box for changing text in a document



So as not to create multiple versions of this dialog box whenever the user chooses the Global Changes command, the application-defined routine `DoGlobalChangesDialog`, shown in Listing 6-20, checks whether the dialog box already exists.

Listing 6-20 Ensuring that the modeless dialog box isn't already open before creating it

```
FUNCTION DoGlobalChangesDialog: OSErr;
BEGIN
   DoGlobalChangesDialog := kSuccess;  {assume success}
   IF gChangeDialogPtr = NIL THEN      {it doesn't exist, so create it}
   BEGIN
      gChangeDialogPtr := GetNewDialog(kGlobalChangesDlog, NIL, Pointer(-1));
      IF gChangeDialogPtr = NIL THEN   {handle failure}
      BEGIN
         DoGlobalChangesDialog := kFailed;
         EXIT(DoShowModelessFindDialogBox);
      END;
      {set window refCon to store value that identifies the dbox}
      SetWRefCon(gChangeDialogPtr, LongInt(kGlobalChangesDlog));
   END
   ELSE        {it does exist, so display and select it}
   BEGIN
      ShowWindow(gChangeDialogPtr); {it's hidden; so show it}
      SelectWindow(gChangeDialogPtr);{bring it to the front}
   END;
   MyAdjustMenus;                {adjust the menus}
END;
```

In this example, a pointer to the modeless dialog box is stored in a global variable. If the global variable does not contain a pointer, `DoGlobalChangesDialog` uses `GetNewDialog` to create and draw the dialog box. Later, if the user decides to close the modeless dialog box, the application merely hides it so that when the user needs it again, `DoGlobalChangesDialog` can display the dialog box in the same location and with the same text selected as when the user last used it. Hiding this dialog box is illustrated later in Listing 6-30 on page 6-94.

If the dialog box has already been created, `DoGlobalChangesDialog` uses the Window Manager procedures `ShowWindow` to make the dialog box visible and `SelectWindow` to make it active.

Finally, `DoGlobalChangesDialog` uses the application-defined routine `MyAdjustMenus` to adjust the menus as appropriate for the modeless dialog box.

Listing 6-34 on page 6-98 illustrates an application-defined routine, `DoActivateGlobalChangesDialog`, that handles activate events for this modeless dialog box. The `DoActivateGlobalChangesDialog` routine in turn uses `DialogSelect`, which sets the graphics port to the modeless dialog box whenever the user makes it active.

Adjusting Menus for Modal Dialog Boxes

The Dialog Manager and the Menu Manager interact to provide various degrees of access to the menus in your menu bar. For alert boxes and modal dialog boxes without editable text items, you can simply allow system software to provide the appropriate access to your menu bar.

When your application displays either an alert box or a modal dialog box (that is, a window of type `dBoxProc`), these actions occur:

1. System software disables all menu items in the Help menu, except the Show Balloons (or Hide Balloons) command, which system software enables.
2. System software disables all menu items in the Application menu.
3. If the Keyboard menu appears in the menu bar, system software enables that menu but disables the About Keyboards command.

When your application displays an alert box or calls the `ModalDialog` procedure for a modal dialog box (described in "Responding to Events in Modal Dialog Boxes" beginning on page 6-82), the Dialog Manager determines whether any of the following cases is true:

- Your application does not have an Apple menu.
- Your application has an Apple menu, but the menu is disabled when the dialog box is displayed.
- Your application has an Apple menu, but the first item in that menu is disabled when the dialog box is displayed.

If none of these cases is true, system software behaves as follows:

1. The Menu Manager disables all of your application's menus.
2. If the modal dialog box contains a visible and active editable text field—and if the menu bar contains a menu having commands with the standard keyboard equivalents Command-X, Command-C, and Command-V—then the Menu Manager enables those three commands and the menu that contains them. The user can then use either the menu commands or their keyboard equivalents to cut, copy, and paste text. (The menu item having keyboard equivalent Command-X must be one of the first five menu items.)

When your application displays alert boxes and modal dialog boxes with no editable text items, it can safely allow system software to handle menu bar access as described in steps 1 and 2.

However, because system software cannot handle the Undo or Clear command (or any other context-appropriate command) for you, your application should handle its own menu bar access for modal dialog boxes with editable text items by performing the following tasks:

- disable the Apple menu or the first item in the Apple menu (typically, your application's About command) in order to take control of its menu bar access when displaying a modal dialog box

- disable all of its menus except the Edit menu, as well as any inappropriate commands in the Edit menu
- use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items
- provide your own code for supporting the Undo command
- enable your application's items in the Help menu as appropriate (system software disables all items except the Hide Balloons / Show Balloons command)

You don't need to do anything else for the system-handled menus—namely, Application, Keyboard, and Help. System software handles these menus for you automatically.

The `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` procedures are described beginning on page 6-132. Your application can test whether a dialog box is the front window when handling mouse-down events in the Edit menu and then call these routines as appropriate.

Figure 6-38 illustrates how an application disables all of its own menus except its Edit menu when displaying a modal dialog box containing editable text items. Access to the Edit menu benefits the user who instead of typing prefers copying from and pasting into editable text items.

Figure 6-38 Menu access when displaying a modal dialog box

Listing 6-21 on the next page shows an application-defined routine, `MyAdjustMenus`, that the SurfWriter application calls to adjust its menus after it displays a window or dialog box, but before it calls `ModalDialog` to handle events in a modal dialog box. When `MyAdjustMenus` determines that the frontmost window is a modal dialog box containing an editable text item, it calls another application-defined routine, `MyAdjustMenusForDialogs`, which adjusts the menus appropriately. Listing 6-22 on the next page shows the `MyAdjustMenusForDialogs` routine.

Listing 6-21 Adjusting menus for various windows

```
PROCEDURE MyAdjustMenus;
VAR
   window:     WindowPtr;
   windowType: Integer;
   menu:       MenuHandle;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:   {document window is in front}
      MyAdjustMenusForDocWindows;
   kMyDialogWindow:  {a dialog box is in front}
      MyAdjustMenusForDialogs;
   kDAWindow:  {adjust menus accordingly for a DA window}
      MyAdjustMenusForDA;
   kNil: {there isn't a front window}
      MyAdjustMenusNoWindows;
   END; {of CASE}
   DrawMenuBar;   {redraw menu bar}
END;
```

The `MyAdjustMenusForDialogs` routine in Listing 6-22 first determines what type of dialog box is in front: modal, movable modal, or modeless. For modal dialog boxes, `MyAdjustMenusForDialogs` disables the Apple menu so that the application can take control of its menus away from the Dialog Manager. The `MyAdjustMenusForDialogs` routine then uses the Menu Manager routines `GetMenuHandle` and `DisableItem` to disable all other application menus except the Edit menu. (To provide help balloons that explain why these menus are unavailable to the user, `MyAdjustMenusForDialogs` uses the Help Manager procedure `HMSetMenuResID` to reassign help resources to these menus; see the chapter "Help Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information.)

Listing 6-22 Disabling menus for a modal dialog box with editable text items

```
PROCEDURE MyAdjustMenusForDialogs;
   VAR
      window:     WindowPtr;
      windowType: Integer;
      myErr:      OSErr;
      menu:       MenuHandle;
BEGIN
   window := FrontWindow;
```

```
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyModalDialogs:
      BEGIN
         menu := GetMenuHandle(mApple);   {get handle to Apple menu}
         IF menu = NIL THEN
            EXIT(MyAdjustMenusForDialogs);
         DisableItem(menu, 0);   {disable Apple menu to get control of menus}
         myErr := HMSetMenuResID(mFile, kFileHelpID); {set up help balloons}
         menu := GetMenuHandle(mFile);       {get handle to File menu}
         IF menu = NIL THEN
            EXIT(MyAdjustMenusForDialogs);
         DisableItem(menu, 0);      {disable File menu}
         myErr := HMSetMenuResID(mFile, kFileHelpID); {set up help balloons}
         IF myErr <> NoErr THEN
            EXIT(MyAdjustMenusForDialogs);
         menu := GetMenuHandle(mTools);   {get handle to Tools menu}
         IF menu = NIL THEN
            EXIT(MyAdjustMenusForDialogs);
         DisableItem(menu, 0);         {disable Tools menu}
         myErr := HMSetMenuResID(mTools, kToolsHelpID);  {help balloons}
         IF myErr <> NoErr THEN
            EXIT(MyAdjustMenusForDialogs);
         MyAdjustEditMenuForModalDialogs;
      END;        {of kMyModalDialogs CASE}
      kMyGlobalChangesModelessDialog:
         ;  {adjust menus here as needed}
      kMyMovableModalDialog:
         ;  {adjust menus here as follows: }
            { disable all menus except Apple, then }
            { call MyAdjustEditMenuForModalDialogs for editable text items}
   END; {of CASE}
END;
```

To adjust the items in the Edit menu, `MyAdjustMenusForDialogs` calls another application-defined routine, `MyAdjustEditMenuForModalDialogs`, which is shown in Listing 6-23 on the next page. The `MyAdjustEditMenuForModalDialogs` routine uses application-defined code to implement the Undo command; uses the Menu Manager procedure `EnableItem` to enable the Cut, Copy, Paste, and Clear commands when appropriate; and disables the commands that support Edition Manager capabilities. Remember that your application should use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items.

Listing 6-23 Adjusting the Edit menu for a modal dialog box

```
PROCEDURE MyAdjustEditMenuForModalDialogs;
VAR
   window:           WindowPtr;
   menu:             MenuHandle;
   selection, undo:  Boolean;
   offset:           LongInt;
   undoText:         Str255;
BEGIN
   window := FrontWindow;
   menu := GetMenuHandle(mEdit); {get a handle to the Edit menu}
   IF menu = NIL THEN            {add your own error handling}
      EXIT (MyAdjustEditMenuForModalDialogs);
   undo := MyIsLastActionUndoable(undoText);
   IF undo THEN   {if action can be undone}
   BEGIN
      EnableItem(menu, iUndo);
      SetMenuItemText(menu, iUndo, undoText);
   END
   ELSE           {if action can't be undone}
   BEGIN
      SetMenuItemText(menu, iUndo, gCantUndo);
      DisableItem(menu, iUndo);
   END;
   selection := MySelection(window);
   IF selection THEN
   BEGIN    {enable editing items if there's a selection}
      EnableItem(menu, iCut);
      EnableItem(menu, iCopy);
   END
   ELSE
   BEGIN    {disable editing items if there isn't a selection}
      DisableItem(menu, iCut);
      DisableItem(menu, iCopy);
   END;
   IF MyGetScrap(NIL, 'TEXT', offset) > 0 THEN
      EnableItem(menu, iPaste) {enable if something to paste}
   ELSE
      DisableItem(menu, iPaste);{disable if nothing to paste}
   DisableItem(menu, iSelectAll);
   DisableItem(menu, iCreatePublisher);
   DisableItem(menu, iSubscribeTo);
   DisableItem(menu, iPubSubOptions);
   END;
END;
```

See the chapter "Menu Manager" in this book for more information on menus and the menu bar.

When the user dismisses the alert box or modal dialog box, the Menu Manager restores all menus to their state prior to the appearance of the alert or modal dialog box—unless your application handles its own menu bar access, in which case you must restore the menus to their previous states. You can use a routine similar to `MyAdjustMenus`, shown in Listing 6-21 on page 6-70, to adjust the menus appropriately according to the type of window that becomes the frontmost window.

Adjusting Menus for Movable Modal and Modeless Dialog Boxes

Although it always leaves the Help, Keyboard, and Application menus and their commands enabled, system software does nothing else to manage the menu bar when you display movable modal and modeless dialog boxes. Instead, your application should allow or deny access to the rest of your menus as appropriate to the context. For example, if your application displays a modeless dialog box for a search-and-replace command, you should allow access to the Edit menu to assist the user with the editable text items, and you should allow use of the File menu so that the user can open another file to be searched. However, you should disable other menus if their commands cannot be used inside the active modeless dialog box.

When creating a modeless dialog box, your application should perform the following tasks:

- disable only those menus whose commands are invalid in the current context
- if the modeless dialog box includes editable text items, use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items

When your application creates a movable modal dialog box, it should perform the following tasks:

- leave the Apple menu enabled so that the user can open other applications with it
- if your movable modal dialog box contains editable text items, leave the Edit menu enabled but use the Dialog Manager procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands
- disable all of your other menus

Listing 6-21 on page 6-70 shows an application-defined routine, `MyAdjustMenus`, that SurfWriter uses to adjust its menus after it displays a window or dialog box**.** You can use a similar routine to adjust your menus as appropriate given the nature of the active window, movable modal dialog box, or modeless dialog box.

Displaying Multiple Alert and Dialog Boxes

You should generally present the user with only one modal dialog box or alert box at a time. Sometimes, you may need to present a modal dialog box and an alert box on the screen at one time. For example, when the user saves a file with the same name as another file, the Standard File Package displays an alert box on top of the standard file dialog box. The alert box asks the user whether to replace the existing file.

Avoid closing a modal dialog box and immediately displaying another modal dialog box or an alert box in response to a user action. This situation creates a "tunneling modal dialog box" effect that might confuse the user. Missing the content of the previous modal dialog box and unable to return to it, the user has difficulty predicting what will happen next.

However, the user should never see more than one modal dialog and one alert box on the screen simultaneously. You can present multiple simultaneous modeless dialog boxes, just as you can present multiple document windows.

When you remove an alert box or a modal dialog box that overlies the default button of a previous alert box, the Dialog Manager doesn't redraw that button's bold outline. Therefore, you should not use an alert box if you need to display another overlapping alert box or dialog box. Instead, you should create a modal dialog box, and you must provide it with an application-defined item that draws the bold outline around the default button. The `ModalDialog` procedure then causes the item to be redrawn after an update event.

In System 7, the Window Manager automatically dims the window frame of a dialog box when you deactivate it to display an alert box, another modal dialog box, or a window. When you deactivate a dialog box, you should use the Control Manager procedure `HiliteControl` to make the controls of a dialog box inactive. You should also draw the outline of the default button of a deactivated dialog box in gray instead of black. Listing 6-16 on page 6-58 shows an application-defined procedure that draws a gray outline when the default button is inactive; Listing 6-34 on page 6-98 shows how to use `HiliteControl` to make buttons inactive and active in response to activate events for a dialog box.

Displaying Alert and Dialog Boxes From the Background

If you ever need to display an alert box or a modal dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. For example, if your application performs lengthy background tasks such as printing many documents or transferring large amounts of data to other computers, you might wish to inform the user that the operation is completed. In these cases, you should post a notification request to notify the user when the operation is completed. Then the Notification Manager automatically displays an alert box containing whatever message you specify; you do not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user

to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to ask the user to bring your application to the foreground. The user can then respond to your alert box or modal dialog box. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for information about the Notification Manager.

Including Color in Your Alert and Dialog Boxes

On color monitors, the Dialog Manager automatically adds color to your alert and dialog boxes so that they match the colors of the windows, alert boxes, and dialog boxes used by system software. These colors provide aesthetic consistency across all monitors, from black-and-white displays to 8-bit color displays. On a color monitor, for example, the racing stripes in the title bar of a modeless dialog box are gray, the close box and window frame are in color, and the buttons and text are black.

When you create alert and dialog resources, your application's alert and dialog boxes use the system's default colors. With the following exceptions, creating alert and dialog resources is typically all you need to do to provide color for your alert and dialog boxes:

- When you need to include a color version of an icon in an alert box or a dialog box, you must create a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource. Plate 2 at the front of this book shows an alert box that includes a color icon.
- If you use `GetNewDialog` or `NewDialog` to create a dialog box and you need to produce a blended gray color for outlining the inactive (that is, dimmed) default button, you must create a **dialog color table** (`'dctb'`) **resource** with the same resource ID as the dialog resource.

"Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 explains how to create a draw routine that outlines the default button of a dialog box. If you deactivate a dialog box, you should dim its buttons and use gray to draw the outline for the default button. Because `GetNewDialog` and `NewDialog` supply black-and-white graphics ports for dialog boxes, you can create a dialog color table resource for the dialog box to force the Dialog Manager to supply a color graphics port. Then you can use a blended gray color for the outline for the default button. (`NewColorDialog` supplies a color graphics port.)

Even when you create a dialog color table resource for drawing a gray outline, you should not change the system's default colors. Listing 6-24 shows a dialog color table resource that leaves the default colors intact but forces the Dialog Manager to supply a color graphics port.

Listing 6-24 Rez input for a dialog color table resource using the system's default colors

```
data 'dctb' (kGlobalChangesDialog, purgeable) {
   $"0000 0000 0000 FFFF"  /*use default colors*/
};
```

By using the system's default colors, you ensure that your application's interface is consistent with that of the Finder and other applications. However, if you feel absolutely compelled to break from this consistency, the Dialog Manager offers you the ability to specify colors other than the default colors. Be aware, however, that nonstandard colors in your alert and dialog boxes may initially confuse your users.

Also be aware that despite any changes you make, users can alter the colors of alert and dialog boxes anyway by changing the settings in the Color control panel.

Your application can specify its own colors in an **alert color table** (`'actb'`) **resource** with the same resource ID as the alert resource or in a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource. Both of these resources have exactly the same format as a window color table (`'wctb'`) resource, described in the chapter "Window Manager" in this book.

▲ **WARNING**

Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not specify colors for these elements if you wish to maintain backward compatibility. ▲

You don't have to call any new routines to change the colors used in alert or dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load a dialog color table resource with the same resource ID as the dialog resource.

Likewise, you can change the system default colors for controls and the color, style, typeface, and size of text used in an alert box or a dialog box by creating an **item color table** (`'ictb'`) **resource** with the same resource ID as the item list resource. You don't have to call any routines to create color items. When you use the `GetNewDialog` function, the Dialog Manager looks first for an item color table resource with the same resource ID as that of the item list resource.

Note

If you want to provide an item color table resource for an alert box or a dialog box, you must create an alert color table resource or a dialog color table resource, even if the item color table resource has no actual color information and describes only static text and editable text style changes. You cannot use an item color table resource to set the font on computers that do not support Color QuickDraw. Also, be aware that changing the default system font makes your application more difficult to localize. ◆

Even if you provide your own `'dctb'`, `'actb'`, or `'ictb'` resources, you do not need to test whether your application is running on a computer that supports Color QuickDraw in order to use these resources.

Handling Events in Alert and Dialog Boxes

The next two sections explain how the Dialog Manager uses the Control Manager to handle events in controls automatically and how it uses TextEdit to handle events in editable text items automatically. The information in these two sections, "Responding to Events in Controls" and "Responding to Events in Editable Text Items," applies to all alert boxes and all types of dialog boxes: modal, modeless, and movable modal.

To display and handle events in alert boxes, you can use the Dialog Manager functions `Alert`, `NoteAlert`, `CautionAlert`, and `StopAlert`. The Dialog Manager handles all of the events generated by the user until the user clicks a button (typically the OK or Cancel button). When the user clicks a button, the alert box functions invert the button that was clicked, close the alert box, and report the user's selection to your application. Your application is responsible for performing the appropriate action associated with that button. This is described in detail in "Responding to Events in Alert Boxes" beginning on page 6-81.

For modal dialog boxes, you use the `ModalDialog` procedure. The Dialog Manager handles most of the user interaction until the user selects an item. The `ModalDialog` procedure then reports that the user selected an enabled item, and your application is responsible for performing the action associated with that item. Your application typically calls `ModalDialog` repeatedly, responding to clicks on enabled items as reported by `ModalDialog`, until the user clicks OK or Cancel. This is described in detail in "Responding to Events in Modal Dialog Boxes" beginning on page 6-82.

For alert boxes and modal dialog boxes, you should also supply an event filter function as one of the parameters to the alert box functions or the `ModalDialog` procedure. As the user interacts with the alert or modal dialog box, these routines pass events to your event filter function before handling each event. Your event filter function can handle any events not handled by the Dialog Manager or, if necessary, can choose to handle events normally handled by the Dialog Manager. This is described in detail in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86.

To handle events in modeless or movable modal dialog boxes, you can use the `IsDialogEvent` function to determine whether the event occurred while a dialog box was the frontmost window. For every type of event that occurs when the dialog box is active (including null events), `IsDialogEvent` returns `TRUE`; otherwise, it returns `FALSE`. When `IsDialogEvent` returns `TRUE`, you can use the `DialogSelect` function to handle key-down events in editable text items automatically, to handle update and activate events automatically, and to report the enabled items that the user clicks. You then respond appropriately to clicks in your active items.

Alternatively, you can handle events in modeless and movable modal dialog boxes much as you handle events in other windows. That is, when you receive an event you can first determine the type of event that occurred and then take the appropriate action according to which window is in front. If a modeless or movable modal dialog box is in front, you can provide code that takes any actions specific to that dialog box and call the `DialogSelect` function to handle any events that your code doesn't handle. The sections "Responding to Mouse Events in Modeless and Movable Modal Dialog Boxes"

beginning on page 6-89, "Responding to Keyboard Events in Modeless and Movable Modal Dialog Boxes" beginning on page 6-94, and "Responding to Activate and Update Events in Modeless and Movable Modal Dialog Boxes" beginning on page 6-97 all take this alternate approach.

Responding to Events in Controls

The Dialog Manager greatly simplifies the work necessary for you to implement buttons, checkboxes, pop-up menus, and radio buttons. For alert boxes and all types of dialog boxes—modal, modeless, and movable modal—the Dialog Manager uses Control Manager routines to display controls automatically, highlight controls appropriately, and report to your application when mouse-down events occur within controls. For example, when the user moves the cursor to an enabled button and holds down the mouse button, the Dialog Manager uses the Control Manager function `TrackControl` to invert the button. When the user releases the mouse button with the enabled button still inverted, the Dialog Manager uses `TrackControl` to report which item was clicked. Your application then responds appropriately—for example, by performing the operation associated with the OK button, by deselecting any other radio button when a radio button is clicked, or by canceling the current operation when the Cancel button is clicked.

For clicks in checkboxes, pop-up menus, and radio buttons, your application usually uses the Control Manager routines `GetControlValue` and `SetControlValue` to get and appropriately set the items' values. The chapter "Control Manager" in this book explains these routines in detail, but this chapter also offers examples of how to use these routines in your alert and dialog boxes. Because the Control Manager does not know how radio buttons are grouped, it doesn't automatically turn one off when the user clicks another one. Instead, it's up to your application to handle this by using the `GetControlValue` and `SetControlValue` routines.

When the user clicks the OK button, your application performs whatever action is necessary according to the values returned by `GetControlValue` for each of the various checkboxes and radio buttons displayed in your alert or dialog box.

When `ModalDialog` and `DialogSelect` call `TrackControl`, they do not allow you to specify any special action procedures necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control that, for example, measures how long the user holds down the mouse button or how far the user has moved an indicator, you can create your own control (or picture or application-defined item that draws a control-like object) in your dialog box. If you use the `ModalDialog` procedure, you must then provide an event filter function that appropriately handles events within that item, and if you use the `DialogSelect` function, you must test for and respond to those events yourself. Alternatively, you can use Window Manager routines to display an appropriate window and then use the Control Manager to create and manage such complex controls yourself. See the chapters "Window Manager" and "Control Manager" in this book for more information.

Responding to Events in Editable Text Items

When the user enters or edits text in an editable text item in your dialog boxes, the Dialog Manager calls TextEdit to handle the events automatically. (You generally shouldn't include editable text items in alert boxes.) You typically disable editable text items because you generally don't need to be informed every time the user types a character or clicks one of them. Instead you need to determine the text only when the OK button is clicked. As illustrated in Listing 6-12 on page 6-49, use `GetDialogItemText` to determine the final value of the editable text item after the user clicks the OK button.

When you use the `ModalDialog` procedure to handle events in modal dialog boxes and when you use the `DialogSelect` function for modeless or movable modal dialog boxes, the Dialog Manager calls TextEdit to handle keystrokes and mouse actions within editable text items, so that

- when the user clicks the item, a blinking vertical bar appears that indicates an insertion point where text may be entered
- when the user drags over text in the item, the text is highlighted; when the user double-clicks a word, the word is highlighted; the highlighted selection is then replaced by what the user types
- when the user holds down the Shift key while clicking or dragging, the highlighted selection is extended or shortened appropriately
- when the user presses the Backspace key, the highlighted selection or the character preceding the insertion point is deleted
- when the user presses the Tab key, the cursor automatically advances to the next editable text item in the item list resource, wrapping around to the first if there are no more items

If your modeless or movable modal dialog box contains any editable text items, call `DialogSelect` even when `WaitNextEvent` returns `FALSE`. This is necessary because the `DialogSelect` function calls the `TEIdle` procedure to make the text cursor blink within your editable text items during null events; otherwise, the text cursor will not blink. Listing 6-25 illustrates an application-defined routine, `DoIdle`, that calls `DialogSelect` whenever the application receives null events while its modeless dialog box is the frontmost window.

Listing 6-25 Using `DialogSelect` during null events

```
PROCEDURE DoIdle (event: EventRecord);
VAR
   window:     WindowPtr;
   windowType: Integer;
   itemHit:    Integer;
   result:     Boolean;
BEGIN
   window := FrontWindow;
   {determine which type of window--document, }
```

```
   { modeless dialog box, etc.--is in front}
   windowType := MyGetWindowType(window);
   CASE windowType OF
   kMyDocWindow:  {document window is frontmost}
      ;  {see examples in "Event Manager" chapter}
   kMyGlobalChangesModelessDialog:  {modeless dialog is frontmost}
      result := DialogSelect(event, window, itemHit);
   END; {of CASE}
END;
```

Generally, your application should handle menu bar access when you display dialog boxes containing editable text items. Leave your Edit menu enabled, and use the `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` procedures to support the Cut, Copy, Paste, and Clear commands and their keyboard equivalents. You should also provide your own code to support the Undo command. "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 describe how to allow users to access your Edit menu when you display dialog boxes.

If you don't supply your own event filter function and the user presses the Return or Enter key while a modal dialog box is onscreen, the Dialog Manager treats the event as a click on the default button (that is, the first item in the list) regardless of whether the dialog box contains an editable text item. If your event filter function responds to the user pressing Return and Enter by moving the cursor in editable text items, don't display a bold outline around any buttons. If your event filter function responds to the user pressing Return and Enter as if the user clicks the default button, then you should display a bold outline around the default button. See "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86 for an example of how to map the Return and Enter keys to the default button in your dialog boxes.

Initially, an editable text item may contain default text or no text. You can provide default text either by specifying a text string as the last element for that item in the item list resource or by using the `SetDialogItemText` procedure, which is described on page 6-131.

When a dialog box that contains editable text items is first displayed, the insertion point usually appears in the first editable text item in the item list resource. You may instead want to use the `SelectDialogItemText` procedure so that the dialog box appears with text selected, or so that an insertion point or a text selection reappears if the user makes an error while entering text. For example, the user who accidentally types nonnumeric input when a number is required can be given the opportunity to type the entry again. The `SelectDialogItemText` procedure is described in detail on page 6-131.

By default, the Dialog Manager displays editable text items in the system font. To maintain visual consistency across applications for your users and to make it easier to localize your application, you should not change the font or font size.

Responding to Events in Alert Boxes

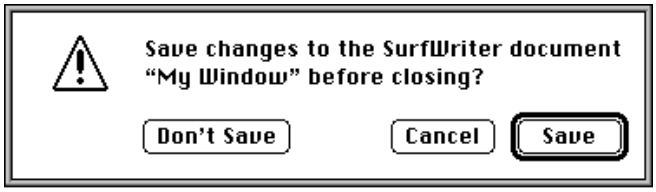
After displaying an alert box or playing an alert sound, the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions call the `ModalDialog` procedure to handle events automatically for you.

The `ModalDialog` procedure, in turn, gets each event by calling the Event Manager function `GetNextEvent`. If the event is a mouse-down event outside the content region of the alert box, `ModalDialog` emits the system alert sound and gets the next event.

The `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions continue calling `ModalDialog` until the user selects an enabled control (typically a button). At this time these functions remove the alert box from the screen and return the item number of the selected control. Your application then responds as appropriate for a click on this item.

For example, the code that supports the alert box displayed in Figure 6-39 must respond to three different events—one for each button that the user may click.

Figure 6-39 Three buttons for which `CautionAlert` reports events



Listing 6-9 on page 6-47 shows an application-defined routine, named `MyCloseDocument`, for the Close command. If the document has been modified since the last save, `MyCloseDocument` displays the alert box illustrated in Figure 6-39 before closing the window. After `MyCloseDocument` displays the caution alert, it tests for the item number that `CautionAlert` returns after it removes the alert box. If the user clicks the Save button, `CautionAlert` returns its item number, and `MyCloseDocument` calls other application-defined routines to save the file, close the file, and close the window. If the user clicks the Don't Save button, `MyCloseDocument` closes the window without saving the file. The only other possible response is for the user to click the Cancel button, in which case `MyCloseDocument` does nothing—the Dialog Manager removes the alert box, and `MyCloseDocument` simply leaves the document window as it is.

The standard event filter function allows users to press the Return or Enter key in lieu of clicking the default button. When one of these keys is pressed, the standard event filter function returns `TRUE` to `ModalDialog`, which in turn causes `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` to return the item number of the default button. When you write your own event filter function, it should emulate the standard filter function by responding in this way to keyboard events involving the Return and Enter keys.

For events inside the alert box, `ModalDialog` passes the event to an event filter function before handling the event. The event filter function provides a secondary event-handling loop for handling events that `ModalDialog` doesn't handle and for overriding events that `ModalDialog` would otherwise handle. You should provide a simple event filter function for every alert box and modal dialog box in your application.

You specify a pointer to your event filter function in the second parameter to the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. In the `MyCloseDocument` routine shown on page 6-47, a pointer to the `MyEventFilter` function is specified. In most cases, you can use the same event filter function in every one of your alert and modal dialog boxes. An example of a simple event filter function that allows background applications to receive update events and performs the other necessary event handling is provided in "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86.

Unless your event filter function handles the event in its own way and returns `TRUE`, `ModalDialog` handles the event inside the alert box as follows:

- In response to an activate or update event for the alert box, `ModalDialog` activates or updates its window.
- If the user presses the mouse button while the cursor is in a control, the Control Manager function `TrackControl` tracks the mouse. If the user releases the mouse button while the cursor is in an enabled control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the control's item number. (Generally, buttons should be the only controls you use in alert boxes.)
- If the user presses the mouse button while the cursor is in any enabled item other than a control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the item number. (Generally, button controls should be the only enabled items in alert boxes.)
- If the user presses the mouse button while the cursor is in a disabled item, or if it is in no item, or if any other event occurs, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` do nothing.

Responding to Events in Modal Dialog Boxes

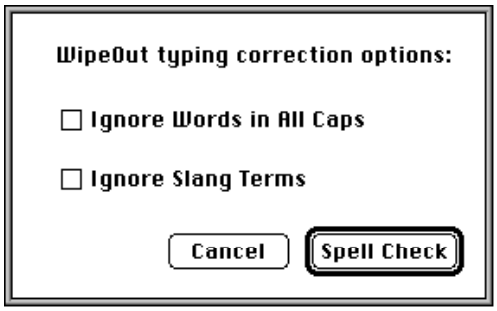
Call the `ModalDialog` procedure immediately after displaying a modal dialog box. This procedure repeatedly handles events inside the modal dialog box until an event involving an enabled item—such as a click in a radio button—occurs. If the event is a mouse-down event outside the content region of the dialog box, `ModalDialog` emits the system alert sound and gets the next event. After receiving an event involving an enabled item, `ModalDialog` returns the item number. Normally you then do whatever is appropriate in response to an event in that item. Your application should continue calling `ModalDialog` until the user selects the OK or Cancel button, at which point your application should close the dialog box.

For example, if the user clicks a radio button, your application should get the value of that button, turn off any other selected radio button within its group, and call `ModalDialog` again to get the next event. If the user clicks the Cancel button, your application should restore the user's work to its state just before the user invoked the dialog box, and then your application should remove the dialog box from the screen.

Note
Do not use `ModalDialog` for modeless or movable modal dialog boxes. ◆

The code that supports the modal dialog box shown in Figure 6-40 must respond to events in four controls: two checkboxes and two buttons.

Figure 6-40 Four items for which `ModalDialog` reports events



Listing 6-26 illustrates an application-defined routine, `MySpellCheckDialog`, that responds to events in these four controls.

Listing 6-26 Responding to events in a modal dialog box

```
FUNCTION MySpellCheckDialog: OSErr;
VAR
   docWindow:           WindowPtr;
   ignoreCapsCheck:     Boolean;
   ignoreSlangCheck:    Boolean;
   spellDialog:         DialogPtr;
   itemHit, itemType:   Integer;
   itemHandle:          Handle;
   itemRect:            Rect;
   capsVal:             Integer;
   slangVal:            Integer;
   event:               EventRecord;
BEGIN
   capsVal := 0;
   slangVal := 0;
   ignoreCapsCheck := FALSE;
   ignoreSlangCheck := FALSE;
   MySpellCheckDialog := kSuccess;{assume success}
   docWindow := FrontWindow;     {get front window}
   IF docWindow <> NIL THEN
```

```
   DoActivate(docWindow, FALSE, event);   {deactivate document window}
spellDialog := GetNewDialog(kSpellCheckID, NIL, Pointer(-1));
IF spellDialog = NIL THEN
BEGIN
   MySpellCheckDialog := kFailed;
   Exit(MySpellCheckDialog);
END;
MyAdjustMenus;             {adjust menus as needed}
GetDialogItem(spellDialog, kUserItem, itemType, itemHandle, itemRect);
SetDialogItem(spellDialog, kUserItem, itemType,
              Handle(@MyDrawDefaultButtonOutline), itemRect);
ShowWindow(spellDialog);   {show dialog box with default button outlined}
REPEAT
   ModalDialog(@MyEventFilter, itemHit);  {get events}
   IF itemHit = kAllCaps THEN    {user clicked Ignore Words in All Caps}
   BEGIN
      {get the control handle to the checkbox}
      GetDialogItem(spellDialog, kAllCaps, itemType, itemHandle,
                    itemRect);
      {get the last value of the checkbox}
      capsVal := GetControlValue(ControlHandle(itemHandle));
      {toggle the value of the checkbox}
      capsVal := 1 - capsVal;
      {set the checkbox to the new value}
      SetControlValue(ControlHandle(itemHandle), capsVal);
   END;
   IF itemHit = kSlang THEN   {user clicked Ignore Slang Terms}
   BEGIN
      {get checkbox's handle, get its value, toggle it, then reset it}
      GetDialogItem(spellDialog, kSlang, itemType, itemHandle, itemRect);
      slangVal := GetControlValue(ControlHandle(itemHandle));
      slangVal := 1 - slangVal;
      SetControlValue(ControlHandle(itemHandle), slangVal);
   END;
UNTIL ((itemHit = kSpellCheck) OR (itemHit = kCancel));
DisposeDialog(spellDialog);      {close the dialog box}
IF itemHit = kSpellCheck THEN    {user clicked Spell Check button}
BEGIN
   IF capsVal = 1 THEN           {user wants to ignore all caps}
      ignoreCapsCheck := TRUE;
   IF slangVal = 1 THEN          {user wants to ignore slang}
      ignoreSlangCheck := TRUE;
```

```
      {now start the spell check}
      SpellCheckMyDoc(ignoreCapsCheck, ignoreSlangcheck);
   END;
END;
```

The `MySpellCheckDialog` routine calls `ModalDialog` immediately after using `GetNewDialog` to create and display the dialog box. The `MySpellCheckDialog` routine repeatedly responds to events in the two checkboxes until the user clicks either the Spell Check or the Cancel button. When the user clicks either of the checkboxes (which are the third and fourth items in the item list resource), `MySpellCheckDialog` uses the `GetDialogItem` procedure to get a handle to the checkbox. The `MySpellCheckDialog` routine coerces this handle to a control handle and passes it to the Control Manager function `GetControlValue` to get the last value of the control (1 if the checkbox was selected or 0 if it was unselected). Subtracting this value from 1, `MySpellCheckDialog` derives a new value for the control. Then `MySpellCheckDialog` passes this value to the Control Manager procedure `SetControlValue` to set the new value. The Control Manager responds by drawing an X in the box if the value of the control is 1 or removing the X if the value of the control is 0.

As soon as the user clicks the Spell Check or Cancel button (which are the first and second items in the item list resource), `MySpellCheckDialog` stops responding to events in the checkboxes. This routine uses the `DisposeDialog` procedure (which is explained in "Closing Dialog Boxes" beginning on page 6-100) to remove the dialog box. If the user clicks the Cancel button, `MySpellCheckDialog` does no further processing of the information in the dialog box. If, however, the user clicks the Spell Check button, `MySpellCheckDialog` calls another application-defined routine, `SpellCheckMyDoc`, to check the document for spelling errors according to the preferences that the user communicated in the checkboxes.

For events inside the dialog box, `ModalDialog` passes the event to an event filter function before handling the event. In this example, the application specifies a pointer to its own event filter function, `MyEventFilter`. As described in the next section, your application should provide an event filter function. You can use the same event filter function in most or all of your alert and modal dialog boxes.

Unless your event filter function handles the event and returns `TRUE`, `ModalDialog` handles the event as follows:

- In response to an activate or update event for the dialog box, `ModalDialog` activates or updates its window.
- If the user presses the mouse button while the cursor is in an editable text item, `ModalDialog` responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If a key-down event occurs and there's an editable text item, text entry and editing are handled as described in "Responding to Events in Editable Text Items" beginning on page 6-79. If the editable text item is enabled, `ModalDialog` returns its item number after it receives either the mouse-down or key-down event. Normally, editable text items are disabled, and you use the `GetDialogItemText` procedure to read the information in the items only after the user clicks the OK button. Listing 6-12 on page 6-49 illustrates this technique.

- If the user presses the mouse button while the cursor is in a control, `ModalDialog` calls the Control Manager function `TrackControl`. If the user releases the mouse button while the cursor is in an enabled control, `ModalDialog` returns the control's item number. Your application should respond appropriately; for example, Listing 6-26 uses an application-defined routine that checks the spelling of a document when the user clicks the Spell Check button.
- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `ModalDialog` returns the item's number, and your application should respond appropriately. Generally, only controls should be enabled. If your application creates a complex control—such as one that measures how far a dial is moved—your application must provide an event filter function to handle mouse events in that item.
- If the user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `ModalDialog` does nothing.

Writing an Event Filter Function for Alert and Modal Dialog Boxes

For alert and modal dialog boxes, the Dialog Manager provides a standard event filter function that checks whether the user has pressed the Enter or Return key and, if so, returns the item number of the default button. In early versions of Macintosh system software, when a single application controlled the computer, the standard event filter function for alert boxes and most modal dialog boxes was usually sufficient. However, because the standard event filter function does not permit background applications to receive or respond to update events, it is no longer sufficient.

Thus, your application should provide a simple event filter function that performs these functions and also allows inactive windows to receive update events. You can use the same event filter function in most or all of your alert and modal dialog boxes.

You can also use your event filter function to handle other events that `ModalDialog` doesn't handle—such as the Command-period key-down event, disk-inserted events, keyboard equivalents, and mouse-down events (if necessary) for application-defined items that you provide.

For example, the standard event filter function ignores key-down events for the Command key. When your application allows the user to access your menus after you display a dialog box, your event filter function should handle keyboard equivalents for menu commands and return `TRUE`.

At a minimum, your event filter function should perform 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 in an application-

defined item. For example, if you provide an application-defined item that requires you to measure how long the user holds down the mouse button or how far the user drags the cursor, use the event filter function to handle events inside that item.

If it seems that you will spend time replicating much of your primary event loop in this event filter function, you might consider handling all the events in your main event loop instead of using the Dialog Manager's `Alert`, `NoteAlert`, `StopAlert`, and `CautionAlert` functions or `ModalDialog` procedure.

Your own event filter function should have three parameters and return a Boolean value. For example, this is how to declare an event filter function named `MyEventFilter`:

```
FUNCTION MyEventFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

After receiving an event that it does not handle, your function should return `FALSE`. When your function returns `FALSE`, `ModalDialog` handles the event, which you pass in the parameter `theEvent`. (Your function can also change the event to simulate a different event and return `FALSE`, which passes the altered event to the Dialog Manager for handling.) If your function does handle the event, your function should return `TRUE` as a function result and, in the `itemHit` parameter, the number of the item that it handled. The `ModalDialog` procedure and, in turn, the `Alert`, `NoteAlert`, `StopAlert`, and `CautionAlert` functions then return this item number in their own `itemHit` parameter.

Because `ModalDialog` calls the `GetNextEvent` function with a mask that excludes disk-inserted events, your event filter function can call the Event Manager procedure `SetSystemEventMask` to accept disk-inserted events. See the chapter "Event Manager" in this book for a discussion about handling disk-inserted events.

For alert and modal dialog boxes, the Dialog Manager provides a standard event filter function that checks whether the user has pressed the Enter or Return key and, if so, returns the item number of the default button. Your event filter function should always check whether the Return key or Enter key was pressed and, if so, return the item number of the default button in the `itemHit` parameter and a function result of `TRUE`. Your event filter function should also check whether the Esc key was pressed and, if so, return the item number for the Cancel button in the `itemHit` parameter and a function result of `TRUE`. Your event filter function should also respond to the Command-period key-down event as if the user had clicked the Cancel button.

To give visual feedback indicating which item has been selected, you should invert buttons that are activated by keyboard equivalents for all alert and dialog boxes. A good rule of thumb is to invert a button for 8 ticks, long enough to be noticeable but not so long as to be annoying. The Control Manager performs this action whenever a user clicks a button, and your application should do this whenever a user presses the keyboard equivalent of a button click.

For modal dialog boxes that contain editable text items, your application should handle menu bar access to allow use of your Edit menu and its Cut, Copy, Paste, Clear, and Undo commands, as explained in "Adjusting Menus for Modal Dialog Boxes" beginning

on page 6-68. Your event filter function should then test for and handle mouse-down events in the menu bar and key-down events for keyboard equivalents of Edit menu commands. Your application should respond to users' choices from the Edit menu by using the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands.

Listing 6-27 shows `MyEventFilter`, which begins by handling update events in windows other than the alert or dialog box. (By responding to update events for your application's own inactive windows in this way, you allow `ModalDialog` to perform a minor switch when necessary so that background applications can update their windows, too.)

Next, `MyEventFilter` handles activate events**.** This event filter function then handles key-down events for the Return and Enter keys as if the user had clicked the default button, and it handles key-down events for the Esc key as if the user had clicked the Cancel button. (See *Inside Macintosh: Text* for information about character codes for the Return, Enter, and Esc keys.) Your event filter function can then include tests for other events, such as disk-inserted events and keyboard equivalents.

Listing 6-27 A typical event filter function for alert and modal dialog boxes

```
FUNCTION MyEventFilter(theDialog: DialogPtr;
                       VAR theEvent: EventRecord;
                       VAR itemHit: Integer): Boolean;
VAR
   key:        Char;
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
   finalTicks: LongInt;
BEGIN
   MyEventFilter := FALSE; {assume Dialog Mgr will handle it}
   IF (theEvent.what = updateEvt) AND
      (WindowPtr(theEvent.message) <> theDialog) THEN
      DoUpdate(WindowPtr(theEvent.message))  {update the window behind}
   ELSE IF (theEvent.what = activateEvt) AND (WindowPtr(theEvent.message)
            <> theDialog) THEN
      DoActivate(WindowPtr(theEvent.message),
                  (BAnd(theEvent.modifiers, activeFlag) <> 0), theEvent)
   ELSE
      CASE theEvent.what OF
         keyDown, autoKey:    {user pressed a key}
         BEGIN
            key := Char(BAnd(theEvent.message, charCodeMask));
            IF (key = Char(kReturnKey)) OR (key = Char(kEnterKey)) THEN
```

```
            BEGIN    {respond as if user clicked Spell Check}
               GetDialogItem(theDialog, kSpellCheck, itemType, itemHandle,
                             itemRect);
                     {invert the Spell Check button for user feedback}
               HiliteControl(ControlHandle(itemHandle), inButton);
               Delay(kVisualDelay, finalTicks); {invert button for 8 ticks}
               HiliteControl(ControlHandle(itemHandle), 0);
               myEventFilter := TRUE;  {event's being handled}
               itemHit := kSpellCheck; {return the default button}
            END;
            IF (key = Char(kEscapeKey)) OR   {user pressed Esc key}
               (Boolean(BAnd(theEvent.modifiers, cmdKey)) AND
               (key = Char(kPeriodKey))) THEN   {user pressed Cmd-pd}
            BEGIN    {handle as if user clicked Cancel}
               GetDialogItem(theDialog, kCancel, itemType, itemHandle,
                             itemRect);
                     {invert the Cancel button for user feedback}
               HiliteControl(ControlHandle(itemHandle), inButton);
               Delay(kVisualDelay, finalTicks); {invert button for 8 ticks}
               HiliteControl(ControlHandle(itemHandle), 0);
               MyEventFilter := TRUE;  {event's being handled}
               itemHit := kCancel;  {return the Cancel button}
            END;  {of Cancel}
            {handle any other keyboard equivalents here}
         END;  {of keydown, autokey}
         {handle disk-inserted and other events here, as needed}
         OTHERWISE
      END;  {of CASE}
END;
```

To use this event filter function for an alert box, the application specifies a pointer to `MyEventFilter` when it calls one of the `Alert` functions, as shown in Listing 6-19 on page 6-66. To use this event filter function for a modal dialog box, the application specifies a pointer to `MyEventFilter` when it calls `ModalDialog`, as shown in Listing 6-26 on page 6-83.

Responding to Mouse Events in Modeless and Movable Modal Dialog Boxes

To handle events in modeless and movable modal dialog boxes, you can use the `IsDialogEvent` function to determine when events occur while a dialog box is the frontmost window. For such events, you can then use the `DialogSelect` function to handle key-down events in editable text items automatically, to handle update and activate events automatically, and to report the enabled items that the user clicks. You must also use additional Toolbox routines to handle other types of keyboard events and other events in the dialog box.

▲ **WARNING**
The `IsDialogEvent` and `DialogSelect` functions are unreliable when running in versions of system software previous to System 7. ▲

Alternatively, and probably most efficiently, your application can respond to events in modeless and movable modal dialog boxes by first determining the type of event that occurred and then taking the appropriate action according to which type of window is in front. If a modeless or movable modal dialog box is in front, you can provide code that takes any actions specific to that dialog box. You can then use the `DialogSelect` function instead of the Control Manager functions `FindControl` and `TrackControl` to handle mouse events in your dialog boxes. The `DialogSelect` function also handles update events, activate events, and events in editable text items. (If your modeless or movable modal dialog box contains editable text items, you should call `DialogSelect` during null events to cause the text cursor to blink.)

If you choose to determine whether events involve movable modal or modeless dialog boxes without the aid of the `IsDialogEvent` function, your application should be prepared to handle the following mouse events:

- clicks in the menu bar, which your application has adjusted as appropriate for the dialog box. Be sure to use the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items in your dialog boxes.
- clicks in the content region of an active movable modal or modeless dialog box. You can use the `DialogSelect` function to aid you in handling the event.
- clicks in the content region of an inactive modeless dialog box. In this case, your application should make the modeless dialog box active by making it the front-most window.
- clicks in the content region of an inactive window whenever a movable modal or modeless dialog box is active. For movable modal dialog boxes, your application should emit the system alert sound, whereas for modeless dialog boxes, your application should bring the inactive window to the front.
- mouse-down events in the drag region (that is, the title bar) of an active movable modal or modeless dialog box. Your application should use the Window Manager procedure `DragWindow` to move the dialog box in response to the user's actions.
- mouse-down events in the drag region of an inactive window when a movable modal dialog box is active. Your application should *not* move the inactive window in response to the user's actions. Instead, your application should play the system alert sound.
- clicks in the close box of a modeless dialog box. Your application should dispose of or hide the modeless dialog box, whichever action is more appropriate.

Figure 6-41 shows a simple modeless dialog box with editable text items.

Listing 6-28 illustrates an application-defined procedure that handles mouse-down events for all windows, including the modeless dialog box shown in Figure 6-41.

Figure 6-41 A modeless dialog box for which `DialogSelect` reports events

Listing 6-28 Handling mouse-down events for all windows

```
PROCEDURE DoMouseDown (event: EventRecord);
VAR
   part:             Integer;
   thisWindow:       WindowPtr;
BEGIN
   {find general location of the cursor at the time of mouse-down event}
   part := FindWindow(event.where, thisWindow);
   CASE part OF   {take action based on the cursor location}
   inMenuBar: ;   {cursor in menu bar; respond with Menu Manager routines}
   inSysWindow: ; {cursor in a DA; use SystemClick here}
   inContent:     {cursor in the content area of one of this app's windows}
      IF thisWindow <> FrontWindow THEN
      BEGIN {mouse-down in a window other than the front }
            { window--make the clicked window the front window, }
            { unless the front window is a movable modal dialog box}
         IF MyIsMovableModal(FrontWindow) THEN
            SysBeep(30)    {emit system alert sound}
         ELSE
            SelectWindow(thisWindow);
      END
      ELSE  {mouse-down in the content area of front window}
         DoContentClick(thisWindow, event);
   inDrag:                 {handle mouse-down in drag area}
      IF (thisWindow <> FrontWindow) AND (MyIsMovableModal(FrontWindow))
      THEN
         SysBeep(30)    {emit system alert sound}
      ELSE
        DragWindow(thisWindow, event.where, GetGrayRgn^^.rgnBBox);
   inGrow: ;            {handle mouse-down in zoom box here}
   inGoAway:            {handle mouse-down in close box here}
      IF TrackGoAway(thisWindow, event.where) THEN
         DoCloseCmd;
   inZoomIn, inZoomOut: ;  {handle zoom box region for standard windows}
   END;     {end of CASE}
END;  {of DoMouseDown}
```

The DoMouseDown routine first uses the Window Manager function FindWindow to determine approximately where the cursor is when the mouse button is pressed. When the user presses the mouse button while the cursor is in the content area of a window, DoMouseDown first checks whether the mouse-down event occurs in the currently active window by comparing the window pointer returned by FindWindow with that returned by the Window Manager function FrontWindow.

When the mouse-down event occurs in an inactive window, DoMouseDown uses another application-defined routine, MyIsMovableModal, to check whether the active window is a movable modal dialog box. If so, DoMouseDown plays the system alert sound. Otherwise, DoMouseDown uses the Window Manager procedure SelectWindow to make the selected window active. (Although not illustrated in this book, the MyIsMovableModal routine uses the Window Manager function GetWVariant to determine whether the variation code for the front window is movableDBoxProc. If so, MyIsMovableModal returns TRUE.) See the chapter "Window Manager" in this book for more information about the SelectWindow and GetWVariant routines.

As in this example, you must ensure that the movable dialog box is modal within your application. That is, the user should not be able to switch to another of your application's windows while the movable modal dialog box is active. Instead, your application should emit the system alert sound. Notice as well that when the mouse-down event occurs in the drag region of any window, DoMouseDown checks whether the drag region belongs to an inactive window while a movable modal dialog box is active. If it does, DoMouseDown again plays the system alert sound. (However, by clicking other applications' windows or by selecting other applications from the Application and Apple menus, users should be able to switch your application to the background when you display a movable modal dialog box—an action users cannot perform with fixed-position modal dialog boxes.)

If a user presses the mouse button while the cursor is in the content region of the active window, DoMouseDown calls another application-defined routine, DoContentClick, to further handle mouse events. Listing 6-29 shows how this routine in turn uses the DialogSelect function to handle the mouse-down event after the application determines that it occurs in the modeless dialog box shown in Figure 6-41 on page 6-91.

Listing 6-29 Using the DialogSelect function for responding to mouse-down events

```
PROCEDURE DoContentClick (thisWindow: windowPtr; event: EventRecord);
VAR
   itemHit:    Integer;
   refCon:     Integer;
BEGIN
   windowType := MyGetWindowType(thisWindow);
   CASE windowType OF
      kMyDocWindow: ;
         {handle clicks in document window here; see the chapter "Control }
         { Manager" for sample code for this case}
```

```
      kGlobalChangesID:    {user clicked Global Changes dialog box}
         BEGIN
            IF DialogSelect(event, DialogPtr(thisWindow), itemHit) THEN
            BEGIN
               IF itemHit = kChange THEN     {user clicked Change}
                  ;  {use GetDialogItem and GetDialogItemText to get }
                     { the text strings and replace one string with the }
                     { other here}
               IF itemHit = kStop THEN    {user clicked Stop}
                  ;  {stop making changes here}
            END;
         END; {of CASE for kGlobalChangesID}
      {handle other window types here}
   END;  {of CASE}
END;
```

In this example, when the user clicks the Change button, `DialogSelect` returns its item number. Within the user's document, the application then performs a global search and replace. (Listing 6-12 on page 6-49 illustrates how an application can use the `GetDialogItem` and `GetDialogItemText` procedures for this purpose.) Generally, only controls should be enabled in a dialog box; therefore, your application normally responds only when `DialogSelect` returns `TRUE` after the user clicks an enabled control. For example, if the event is an activate or update event for a dialog box, `DialogSelect` activates or updates it and returns `FALSE`, so your application does not need to respond to the event.

At this point, you may also want to check for and respond to any special events that you do not wish to pass to `DialogSelect` or that require special processing before you pass them to `DialogSelect`. You would need to do this, for example, if the dialog box needs to respond to disk-inserted events.

IMPORTANT

When `DialogSelect` calls `TrackControl`, it does not allow you to specify any action procedures necessary for a more complex control—for example, a control that measures how long the user holds down the mouse button or one that measures how far the user has moved an indicator. For instances like this, you can create a picture or an application-defined item that draws a control-like object; you must then test for and respond to those events yourself before passing events to `DialogSelect`. Or, you can use the Control Manager functions `FindControl` and `TrackControl` to process the mouse events inside the controls of your dialog box. ▲

Listing 6-28 on page 6-91 calls one of its application-defined routines, `DoCloseCmd`, whenever the user clicks the close box of the active window. If the active window is a modeless dialog box, you might find it more efficient to hide the window rather than remove its data structures. Listing 6-30 shows how you can use the Window Manager routine `HideWindow` to hide the Global Changes modeless dialog box when the user

clicks its close box. The next time the user chooses the Global Changes command, the dialog box is already available, in the same location and with the same text selected as when it was last used. (Listing 6-20 on page 6-67 illustrates how first to create and later redisplay this modeless dialog box.)

Listing 6-30 Hiding a modeless dialog box in response to a Close command

```
PROCEDURE DoCloseCmd;
VAR
   myWindow: WindowPtr;
   myData: MyDocRecHnd;
   windowType: Integer;
BEGIN
   myWindow := FrontWindow;
   windowType := MyGetWindowType(myWindow);
   CASE windowType OF
      kMyGlobalChangesModelessDialog:
         HideWindow(myWindow);
      kMySpellModelessDialog:
         HideWindow(myWindow);
      kMyDocWindow:
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(myWindow));
            MyCloseDocument(myData);
         END;  {of kMyDocWindow case}
      kDAWindow:
         CloseDeskAcc(WindowPeek(myWindow)^.windowKind);
   END; {of CASE}
END;
```

Responding to Keyboard Events in Modeless and Movable Modal Dialog Boxes

If you adopt the previously described strategy of determining—without the aid of the `IsDialogEvent` function—whether events involve movable modal or modeless dialog boxes, your application should be prepared to handle the following keyboard events:

- keyboard equivalents, such as Command-C to copy, to which your application should respond appropriately
- key-down events for the Return and Enter keys, to which your application should respond as if the user had clicked the default button
- key-down events for the Esc or Command-period keystrokes, to which your application should respond as if the user had clicked the Cancel button
- key-down and auto-key events in editable text items, for which your application can use the `DialogSelect` function, which in turn calls TextEdit to handle keystrokes within editable text items automatically

Listing 6-31 illustrates how an application can check for keyboard equivalents whenever it receives key-down events. If the user holds down the Command key while pressing another key, the application calls another of its application-defined procedures, `DoMenuCommand`, which handles keyboard equivalents for menu commands. See the chapter "Menu Manager" in this book for an example of a `DoMenuCommand` procedure. Remember that when a movable modal dialog box or a modeless dialog box is active, your application should adjust the menus appropriately, and use the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands in editable text items.

Listing 6-31 Checking for key-down events involving the Command key

```
PROCEDURE DoKeyDown (event: EventRecord);
VAR
   key:  Char;
BEGIN
   key := CHR(BAnd(event.message, charCodeMask));
   IF BAnd(event.modifiers, cmdKey) <> 0 THEN
   BEGIN                        {Command key down}
      IF event.what = keyDown THEN
      BEGIN
         MyAdjustMenus;             {adjust the menus as needed}
         DoMenuCommand(MenuKey(key));  {handle the menu command}
      END;
   END
   ELSE
      MyHandleKeyDown(event);
END;
```

After determining that a key-down event does not involve a keyboard equivalent, Listing 6-31 calls another of its own routines, `MyHandleKeyDown`, which is shown in Listing 6-32.

Listing 6-32 Checking for key-down events in a modeless dialog box

```
PROCEDURE MyHandleKeyDown (event: EventRecord);
VAR
   window:        WindowPtr;
   windowType:    Integer;
BEGIN
   window := FrontWindow;
   {determine the type of window--document, modeless, etc.}
```

```
      windowType := MyGetWindowType(window);
      IF windowType = kMyDocWindow THEN   {key-down in doc window}
      BEGIN    {handle keystrokes in document window here}
      END
      ELSE     {key-down in modeless dialog box}
         MyHandleKeyDownInModeless(event, windowType);
   END;
```

The `MyHandleKeyDown` routine determines what type of window is active when the user presses a key. If a modeless dialog box is the frontmost window, `MyHandleKeyDown` automatically calls another application-defined routine, `MyHandleKeyDownInModeless`, to respond to key-down events in modeless dialog boxes. The `MyHandleKeyDownInModeless` routine is shown in Listing 6-33.

Listing 6-33 Responding to key-down events in a modeless dialog box

```
PROCEDURE MyHandleKeyDownInModeless(event: EventRecord; windowType: Integer);
VAR
   key:        Char;
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
   finalTicks: LongInt;
   handled:    Boolean;
   item:       Integer;
   theDialog:  DialogPtr;
BEGIN
   handled := FALSE;
   theDialog := FrontWindow;
   CASE windowType OF
      kGlobalChangesID:    {key-down in Global Changes dialog box}
      BEGIN
         key := Char(BAnd(event.message, charCodeMask));
         IF (key = Char(kReturnKey)) OR (key = Char(kEnterKey)) THEN
         BEGIN    {respond as if user clicked Change}
            GetDialogItem(theDialog, kChange, itemType, itemHandle,
                          itemRect);
                {invert the Change button for 8 ticks for user feedback}
            HiliteControl(ControlHandle(itemHandle), inButton);
            Delay(kVisualDelay, finalTicks);
            HiliteControl(ControlHandle(itemHandle), 0);
               {use GetDialogItem and GetDialogItemText to get the text }
               { strings and replace one string with the other here}
            handled := TRUE;  {event's been handled}
         END;
```

```
        IF (key = Char(kEscapeKey)) OR   {user pressed Esc key}
           (Boolean(BAnd(event.modifiers, cmdKey)) AND
              (key = Char(kPeriodKey))) THEN   {user typed Cmd-pd}
        BEGIN    {handle as if user clicked Stop}
           GetDialogItem(theDialog, kStop, itemType, itemHandle,
                         itemRect);
                {invert the Stop button for 8 ticks for user feedback}
           HiliteControl(ControlHandle(itemHandle), inButton);
           Delay(kVisualDelay, finalTicks);
           HiliteControl(ControlHandle(itemHandle), 0);
              {cancel the current operation here}
           handled := TRUE;  {event's been handled}
        END;
        IF NOT handled THEN  {let DialogSelect handle keydown events in }
                             { editable text items}
           handled := DialogSelect(event, theDialog, item);
     END;  {of case kGlobalChangesID}
  {handle other modeless and movable modal dialog boxes here}
  END;  {of CASE}
END;
```

When `MyHandleKeyDownInModeless` determines that the front window is the Global Changes modeless dialog box, it checks whether the user pressed Return or Enter. If so, `MyHandleKeyDownInModeless` responds as if the user had clicked the default button: Change. The `MyHandleKeyDownInModeless` routine uses the Control Manager procedure `HiliteControl` to highlight the Change button for 8 ticks. (Listing 6-27 on page 6-88 illustrates how to use `HiliteControl` to highlight the button from within a modal dialog box's event filter function.)

When the user presses Esc or Command-period, `MyHandleKeyDownInModeless` responds as if the user had clicked the Cancel button.

Finally, `MyHandleKeyDownInModeless` uses the `DialogSelect` function, which in turn calls TextEdit to handle keystrokes within editable text items.

Responding to Activate and Update Events in Modeless and Movable Modal Dialog Boxes

If you adopt the previously described strategy of determining—without the aid of the `IsDialogEvent` function—whether events involve movable modal or modeless dialog boxes, your application should be prepared to handle activate and update events for both movable modal and modeless dialog boxes. You can use `DialogSelect` to assist you in handling activate and update events. For faster performance, you may instead want to use the `UpdateDialog` function when handling update events. Both `DialogSelect` and `UpdateDialog` use the QuickDraw procedure `SetPort` to make the dialog box the current graphics port before redrawing or updating it.

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 a modeless or movable modal dialog box again, you should use `HiliteControl` to make the controls active again.

The application-defined `DoActivateGlobalChangesDialog` routine shown in Listing 6-34 illustrates how to use `HiliteControl` to make the Change button active when activating a modeless dialog box and how to make the Change and Stop buttons inactive when deactivating the dialog box.

Listing 6-34 Activating a modeless dialog box

```
PROCEDURE DoActivateGlobalChangesDialog (window: WindowPtr;
                                         event: EventRecord);
VAR
   activate:   Boolean;
   handled:    Boolean;
   item:       Integer;
   itemType:   Integer;
   itemHandle: Handle;
   itemRect:   Rect;
BEGIN
   MyCheckEvent(event); {get a valid event record to pass to DialogSelect}
   activate := (BAnd(event.modifiers, activeFlag) <> 0);
   IF activate THEN     {activate the modeless dialog box}
   BEGIN
      {highlight editable text}
      SelectDialogItemText(window, kFindText, 0, 32767);
      {make the Change button active (make the Stop button active }
      { only during a change operation)}
      GetDialogItem(DialogPtr(window), kChange, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 0); {make Change active}
      {draw a bold outline around the newly activated Change button}
      MyDrawDefaultButtonOutline(DialogPtr(window), kChange);
   END
   ELSE     {dim the Change and Stop buttons for a deactivate dialog box}
   BEGIN
      GetDialogItem(DialogPtr(window), kChange, itemType, itemHandle,
                    itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);  {dim Change button}
```

```
      {draw a gray outline around the newly dimmed Change button}
      MyDrawDefaultButtonOutline(DialogPtr(window), kChange);
      GetDialogItem(DialogPtr(window), kStop, itemType, itemHandle,
                  itemRect);
      HiliteControl(ControlHandle(itemHandle), 255);  {dim Stop button}
   END;
   {let Dialog Manager handle activate events}
   handled := DialogSelect(event, window, item);
   MyAdjustMenus;    {adjust the menus appropriately}
END;
```

The `DoActivateGlobalChangesDialog` routine uses `DialogSelect` to handle activate events in the modeless dialog box. In response to an activate event, `DialogSelect` handles the event and returns `FALSE`. The `DialogSelect` function sets the current graphics port to the modeless dialog box whenever the user makes it active.

Because `DialogSelect` expects three parameters, one of which must be an event record, `DoActivateGlobalChangesDialog` uses the application-defined routine `MyCheckEvent` to verify that the event is a valid event. If it's not, `MyCheckEvent` creates and returns a valid event record for an activate event.

Because `DialogSelect` doesn't call any draw procedures for items in response to activate events, `DoActivateGlobalChangesDialog` calls the application-defined draw routine `MyDrawDefaultButtonOutline` to draw either a black outline around the default button when activating the dialog box or a gray outline when deactivating it. The `MyDrawDefaultButtonOutline` routine is shown in Listing 6-17 on page 6-59.

Because users can switch out of your application when you display a movable modal dialog box, your application must handle activate events for it, too.

You can also use `DialogSelect` to handle update events. In response to an update event, `DialogSelect` calls the Window Manager procedure `BeginUpdate`, the Dialog Manager procedure `DrawDialog` to redraw the entire dialog box, and then the Window Manager procedure `EndUpdate`. However, a faster way to update the dialog box is to use the `UpdateDialog` procedure, which redraws only the update region of a dialog box. As shown in Listing 6-35, you should call `BeginUpdate` before using `UpdateDialog`, and then call `EndUpdate`.

Listing 6-35 Updating a modeless dialog box

```
PROCEDURE DoUpdate (window: WindowPtr);
VAR
      windowType: Integer;
BEGIN
   windowType := MyGetWindowType(window);
   CASE windowType OF
      kMyDocWindow:
         ;  {update document windows here}
```

```
      kMyGlobalChangesModelessDialog:
         BEGIN
            BeginUpdate(window);
            UpdateDialog(window, window^.visRgn);
            EndUpdate(window);
         END;
      {handle cases for other window types here}
   END; {of CASE}
END;
```

Closing Dialog Boxes

When you no longer need a dialog box, you can dispose of it by using either the `CloseDialog` procedure if you allocated the memory for the dialog box or the `DisposeDialog` procedure if you did not. Or, you can merely make it invisible by using the Window Manager procedure `HideWindow`.

Generally, your application should not allocate memory for modal dialog boxes or movable modal dialog boxes, but it should allocate memory for modeless dialog boxes. Under these circumstances, your application should use `DisposeDialog` to dispose of either a fixed or movable modal dialog box when the user clicks the OK or Cancel button, and it should use `CloseDialog` to dispose of a modeless dialog box when the user clicks the close box or chooses Close from the File menu.

You do not close alert boxes; the Dialog Manager does that for you automatically by calling the `DisposeDialog` procedure after the user responds to the alert box by clicking any enabled button.

The `CloseDialog` procedure removes a dialog box from the screen and deletes it from the window list. It also releases the memory occupied by

- the data structures associated with the dialog box (such as its structure, content, and update regions)
- all the items in the dialog box (except for pictures and icons, which might be shared by other resources) and any data structures associated with them—for example, the region occupied by the scroll box of a scroll bar

The `CloseDialog` procedure does not dispose of the dialog record or the item list resource. Unlike `GetNewDialog`, `NewDialog` does not use a copy of the item list resource. So, if you create a dialog box with `NewDialog`, you may want to use `CloseDialog` to keep the item list resource in memory even if you didn't supply a pointer to the memory.

The `DisposeDialog` procedure calls `CloseDialog` and, in addition, releases the memory occupied by the dialog's item list resource and the dialog record. If you passed `NIL` as a parameter to `GetNewDialog` or `NewDialog` to let the Dialog Manager allocate memory in the heap, call `DisposeDialog` when you're done with a dialog box.

For modeless and movable modal dialog boxes, you might find it more efficient to hide the dialog box rather than remove its data structures. Listing 6-30 on page 6-94 uses the Window Manager routine `HideWindow` to hide the Global Changes modeless dialog box

when the user clicks its close box. The next time the user invokes the Global Changes command, the dialog box is already available, in the same location and with the same text selected as when it was last used.

If you adjust the menus when you display a dialog box, be sure to return them to an appropriate state when you close the dialog box, as described in "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73.

Dialog Manager Reference

This section describes the data structure, routines, and resources that are specific to the Dialog Manager.

The "Data Structure" section shows the Pascal data structure for the dialog record, which the Dialog Manager creates and maintains. The "Dialog Manager Routines" section describes Dialog Manager routines for invoking alerts, creating and disposing of dialog boxes, manipulating items in alert and dialog boxes, and handling events in dialog boxes.

The "Application-Defined Routines" section describes routines that your application must supply when you need to create application-defined items in dialog boxes, to filter events that the Dialog Manager doesn't handle, and to define its own alert sounds.

The "Resources" section describes the dialog resource, the alert resource, the item list resource, the dialog color table resource, the alert color table resource, and the item color table resource. The summary sections that conclude this chapter include listings of the constants that define values for the item types in alert and dialog boxes, the OK and Cancel buttons in alert boxes, and the icons in note alert boxes, caution alert boxes, and stop alert boxes, along with the constants used by the `Gestalt` function for the Dialog Manager.

Data Structure

This section describes the dialog record. Your application doesn't need to create or use this record; rather, your application simply uses the appropriate Dialog Manager routines, creates any necessary resources, and then allows the Dialog Manager to create and use records of this data type as necessary. The dialog record is described here for completeness only.

The Dialog Record

To create an alert or a dialog box, you use a Dialog Manager routine—such as `Alert` or `GetNewDialog`—that incorporates information from your item list resource and from your alert resource or dialog resource into a data structure, called a *dialog record*, in memory. The Dialog Manager creates a dialog record, which is a data structure of type `DialogRecord`, whenever your application creates an alert or a dialog box. Your application generally should not create a dialog record or directly access its fields.

```
TYPE  DialogPtr      = WindowPtr;
      DialogPeek     = ^DialogRecord

      DialogRecord   =
      RECORD
         window:     WindowRecord;  {dialog window}
         items:      Handle;        {item list resource}
         textH:      TEHandle;      {current editable text item}
         editField:  Integer;       {editable text item number }
                                    { minus 1}
         editOpen:   Integer;       {used internally; reserved}
         aDefItem:   Integer;       {default button item number}
      END;
```

Field descriptions

`window`	The window record for the alert box or dialog box.
`items`	A handle to the item list resource for the alert or the dialog box.
`textH`	A handle to the current editable text item.
`editField`	The current editable text item.
`editOpen`	Used internally; reserved.
`aDefItem`	The item number of the default button.

Dialog Manager Routines

This section describes the routines for initializing the Dialog Manager, invoking alerts, creating and disposing of dialog boxes, manipulating items in alert and dialog boxes, and handling events in alert and dialog boxes.

Some Dialog Manager routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, `GetDialogItem` is also available as `GetDItem`. Table 6-1 provides a mapping between the previous name of a routine and its new equivalent name.

Table 6-1 Mapping between new and previous names of Dialog Manager routines

New name	Previous name
`DialogCopy`	`DlgCopy`
`DialogCut`	`DlgCut`
`DialogDelete`	`DlgDelete`
`DialogPaste`	`DlgPaste`
`DisposeDialog`	`DisposDialog`
`FindDialogItem`	`FindDItem`

Table 6-1 Mapping between new and previous names of Dialog Manager routines (continued)

New name	Previous name
GetAlertStage	GetAlrtStage
GetDialogItem	GetDItem
GetDialogItemText	GetIText
HideDialogItem	HideDItem
NewColorDialog	NewCDialog
ResetAlertStage	ResetAlrtStage
SelectDialogItemText	SelIText
SetDialogFont	SetDAFont
SetDialogItem	SetDItem
SetDialogItemText	SetIText
ShowDialogItem	ShowDItem
UpdateDialog	UpdtDialog

Initializing the Dialog Manager

Before using the Dialog Manager, you must initialize—in order—QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit. The first Dialog Manager routine to call is the `InitDialogs` procedure, which initializes the Dialog Manager.

At your application's request, the Dialog Manager uses the system alert sound for signaling the user during various alert stages. For alerts, if you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure (described on page 6-144) and call the `ErrorSound` procedure to make it the current sound procedure.

By default, the Dialog Manager displays static text and editable text items in the system font. To make it easier to localize your application for use with worldwide versions of system software, you should not change the font. However, if you determine that it is imperative for your application to display static text or editable text in a font other than the system font, you can use the `SetDialogFont` procedure.

InitDialogs

Use the `InitDialogs` procedure to initialize the Dialog Manager.

```
PROCEDURE InitDialogs (resumeProc:  ResumeProcPtr);
```

`resumeProc`
A pointer to a procedure used by the System Error Handler in case a fatal system error occurs on a system that predates MultiFinder. For System 7, your application should set this parameter to `NIL`.

DESCRIPTION

Before using the Dialog Manager, you must initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit, in that order. Then, to initialize the Dialog Manager, call `InitDialogs` once before all other Dialog Manager routines. The `InitDialogs` procedure does the following initialization:

- It saves the pointer passed in the `resumeProc` parameter. For System 7, your application should set the `resumeProc` parameter to `NIL`.
- It installs the system alert sound. To change the system alert sound, use the `ErrorSound` procedure.
- It passes empty strings to the `ParamText` procedure.

ErrorSound

To use your own alert sound instead of the system alert sound for signaling the user, use the `ErrorSound` procedure.

```
PROCEDURE ErrorSound (soundProc: SoundProcPtr);
```

`soundProc` A pointer to a procedure that generates the desired alert sounds.

DESCRIPTION

The Dialog Manager uses the system alert sound for signaling the user during various alert stages. The system alert sound, which is a sound resource stored in the System file, is played whenever system software or your application uses the Sound Manager procedure `SysBeep`. By changing the setting in the Sound control panel, the user can determine which sound is played. If you want to use sounds other than the system alert sound at various alert stages, write your own sound procedure and call the `ErrorSound` procedure to make it the current sound procedure.

SPECIAL CONSIDERATIONS

If you pass `NIL` in the `soundProc` parameter, the Dialog Manager neither plays sounds nor causes the menu bar to blink, and thus the user receives no signal.

SEE ALSO

See the description of `MyAlertSound` on page 6-144 for a discussion of how to write the sound procedure pointed to by the `soundProc` parameter. For examples of how to incorporate sound alerts into alert stages, see Listing 6-2 on page 6-21 and Listing 6-3 on page 6-22.

SetDialogFont

Although you generally should not change the font used in static and editable text items, you can do so with the `SetDialogFont` procedure. The `SetDialogFont` procedure is also available as the `SetDAFont` procedure.

```
PROCEDURE SetDialogFont (fontNum: Integer);
```

`fontNum` A font ID number. Do not rely on font number constants. Instead, use the Font Manager function `GetFNum` to find the font number to pass in this parameter.

DESCRIPTION

For subsequently created dialog and alert boxes, `SetDialogFont` sets the font of the dialog or alert box's graphics port to the specified font. If you don't call this procedure, the system font is used. The `SetDialogFont` procedure does not affect titles of controls, which are always displayed in the system font.

SPECIAL CONSIDERATIONS

There are a number of caveats regarding the `SetDialogFont` procedure.

First, the Standard File Package does not always properly calculate the position of the standard file dialog box once this procedure has been called; for example, the standard file dialog box may be partially obscured by a menu bar. Second, be aware that this procedure affects all static text and editable text items in all of the alert and dialog boxes you subsequently display. Third, `SetDialogFont` does not change the font for control titles. Fourth, you can't use `SetDialogFont` to change the font size or font style. Finally, and most importantly, your application will be much easier to localize if you always use the system font in your alert and dialog boxes and never use `SetDialogFont`.

SEE ALSO

See the chapter "Font Manager" in *Inside Macintosh: Text* for information about the `GetFNum` function.

Creating Alerts

To create an alert—consisting of an alert sound, an alert box, or both—use one of these functions: `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert`. The first three functions display, respectively, the note, caution, and stop alert icons (see Figure 6-3, Figure 6-4, and Figure 6-5) in the upper-left corner of the alert box. The `Alert` function allows you to display your own icon or to have no icon at all in the upper-left corner of your alert box.

These functions take descriptive information about the alert from an alert resource that you provide. 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 a dialog record, play an alert sound, and display an alert box according to the alert stages that you specify in the alert resource.

You should specify a pointer to an event filter function when you call the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. You can use the same event filter function in most or all of your alert and modal dialog boxes.

If you need to find out the current alert stage—for example, to ensure that your application deactivates the frontmost window only if an alert box is to be displayed at that stage—use the `GetAlertStage` function. To change the current alert stage, use the `ResetAlertStage` procedure.

Your application does not dispose of alert boxes; the Dialog Manager does that for you automatically.

Alert

To display an alert box (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), you can use the `Alert` function. This function does not display a default icon in the upper-left corner of the alert box; you can leave this area blank, or you can specify your own icon in the alert's item list resource, which in turn is specified in the alert resource.

```
FUNCTION Alert (alertID: Integer;
                filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure.

DESCRIPTION

The `Alert` function creates the alert defined in the specified alert resource. The function calls the current alert sound procedure and passes it the sound number specified in the alert resource for the current alert stage. If no alert box is to be drawn at this stage, `Alert` returns –1; otherwise, it uses the `NewDialog` function to create and display the alert box. The default system window colors are used unless your application provides an alert color table resource with the same resource ID as the alert resource.

The `Alert` function uses the `ModalDialog` procedure, which repeatedly gets and handles most events for you. The `ModalDialog` procedure, in turn, gets each event by calling the Event Manager function `GetNextEvent`. If the event is a mouse-down event outside the content region of the alert box, `ModalDialog` emits an error sound and gets the next event.

The `Alert` function continues calling `ModalDialog` until the user selects an enabled control (typically a button), at which time the `Alert` function removes the alert box from the screen and returns the item number of the selected control. Your application then responds as appropriate when the user clicks this item.

For events inside the alert box, `ModalDialog` passes the event to an event filter function before handling the event. The event filter function provides a secondary event-handling loop for events that `ModalDialog` doesn't handle. You specify a pointer to your event filter function in the `filterProc` parameter of the `Alert` function.

If you set the `filterProc` parameter to `NIL`, the Dialog Manager uses the standard event filter function, which behaves as follows:

- If the user presses the Return or Enter key, the event filter function returns `TRUE` and returns the item number for the default button.

However, your application should provide a simple event filter function that not only replicates this behavior but also

- returns `TRUE` and the item number for the Cancel button if the user presses Esc or Command-period
- updates your windows in response to update events (this also allows background windows to receive update events) and returns `FALSE`
- returns `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.

Unless the event filter function handles the event in its own way and returns `TRUE`, `ModalDialog` handles the event inside the alert box as follows:

- If the user presses the mouse button while the cursor is in a control, the Control Manager function `TrackControl` tracks the cursor. If the user releases the mouse button while the cursor is in an enabled control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the control's item number. (Generally, buttons should be the only controls you use in alert boxes.)
- If the user presses the mouse button while the cursor is in any enabled item other than a control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the item number. (Generally, button controls should be the only enabled items in alert boxes.)
- If user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` do nothing.

The `Alert` function uses the QuickDraw routine `SetPort` to make the alert box the current graphics port. It's not necessary for your application to call `SetPort` again before displaying alert boxes, because you can't draw into any other windows between the time you create an alert box and the time the Dialog Manager displays it.

SPECIAL CONSIDERATIONS

If you need to display an alert box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you will not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to an alert box that your application presents.

SEE ALSO

The `ModalDialog` procedure is described on page 6-135. See "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86 for a discussion of how to write an event filter function. See "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 for a discussion of alerts and alert stages. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about button titles and messages in alert boxes. Alert resources are described on page 6-150. Alert color table resources are described on page 6-157. The Dialog Manager uses the system alert sound as the error sound unless you change it by calling the `ErrorSound` procedure, described on page 6-104. See "Responding to Events in Alert Boxes" beginning on page 6-81 for a discussion of how to respond to events returned by the `Alert` function. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for information about the Notification Manager.

The `NoteAlert`, `CautionAlert`, and `StopAlert` functions are identical to the `Alert` function, except that `NoteAlert` (described on page 6-110), `CautionAlert` (described on page 6-111), and `StopAlert` (described next) display icons in the upper-left corners of alert boxes.

StopAlert

To display an alert box with a stop icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `StopAlert` function.

```
FUNCTION StopAlert (alertID: Integer;
                    filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `StopAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `StopAlert` draws the stop icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The stop icon has the following resource ID:

```
CONST stopIcon = 0;  {stop icon}
```

By default, the Dialog Manager uses the standard stop icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

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

SEE ALSO

Figure 6-5 on page 6-9 illustrates the stop icon in a typical stop alert. Except that it includes a stop icon in the alert box, `StopAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

NoteAlert

To display an alert box with a note icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `NoteAlert` function.

```
FUNCTION NoteAlert (alertID: Integer;
                    filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `NoteAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `NoteAlert` draws the note icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The note icon has the following resource ID:

```
CONST noteIcon = 1;  {note icon}
```

By default, the Dialog Manager uses the standard note icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a note alert to inform users of a minor mistake 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 an OK button. Occasionally, a note alert may ask a simple question and provide a choice of responses.

SEE ALSO

Figure 6-3 on page 6-8 illustrates the note icon in a typical note alert. Except that it includes a note icon in the alert box, `NoteAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

CautionAlert

To display an alert box with a caution icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `CautionAlert` function.

```
FUNCTION CautionAlert (alertID: Integer;
                       filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `CautionAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `CautionAlert` draws the caution icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The caution icon has the following resource ID:

```
CONST cautionIcon = 2;  {caution icon}
```

By default, the Dialog Manager uses the standard caution icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a caution alert to alert the user of an operation that may have undesirable results if it's allowed to continue. Give the user the choice of continuing the action (by clicking an OK button) or stopping it (by clicking a Cancel button).

SEE ALSO

Figure 6-4 on page 6-9 illustrates the caution icon in a typical caution alert. Except that it includes a caution icon in the alert box, `CautionAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

GetAlertStage

To determine the stage of the last occurrence of an alert, use the `GetAlertStage` function. The `GetAlertStage` function is also available as the `GetAlrtStage` function.

```
FUNCTION GetAlertStage: Integer;
```

DESCRIPTION

The `GetAlertStage` function returns a number from 0 to 3 as the stage of the last occurrence of an alert. For example, you can use the `GetAlertStage` function to ensure that your application deactivates the active window only if an alert box is to be displayed at that stage.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ACount` contains this number. In addition, the global variable `ANumber` contains the resource ID of the alert resource of the last alert that occurred.

SEE ALSO

Listing 6-19 on page 6-66 illustrates how to use `GetAlertStage` to determine whether to deactivate a window for the current alert stage. Listing 6-2 on page 6-21 illustrates how to use an alert resource to specify different alert responses according to different alert stages.

ResetAlertStage

To reset the current alert stage to the first alert stage, use the `ResetAlertStage` procedure. The `ResetAlertStage` procedure is also available as the `ResetAlrtStage` procedure.

```
PROCEDURE ResetAlertStage;
```

DESCRIPTION

The `ResetAlertStage` procedure resets every alert to a first-stage alert.

SEE ALSO

Listing 6-2 on page 6-21 illustrates how to use an alert resource to specify different alert responses according to different alert stages.

Creating and Disposing of Dialog Boxes

To create a dialog box, you should generally use the `GetNewDialog` function, which takes information about the dialog from a dialog 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. However, you can also use the `NewDialog` and `NewColorDialog` functions—for which you pass descriptive information in parameters—to create dialog boxes.

The `NewColorDialog` function is identical to the `NewDialog` function, except that `NewColorDialog` returns a pointer to a color graphics port.

When you no longer need a dialog box, use the `CloseDialog` procedure if you allocated the memory for the dialog record of the dialog box and use the `DisposeDialog` procedure if you did not. (To merely make the dialog box invisible to the user, you can use the Window Manager procedure `HideWindow`.)

GetNewDialog

To create a dialog box from a description in a dialog resource, use the `GetNewDialog` function.

```
FUNCTION GetNewDialog (dialogID: Integer; dStorage: Ptr;
                       behind: WindowPtr): DialogPtr;
```

`dialogID` The resource ID of a dialog resource. If the dialog resource is missing, the Dialog Manager returns to your application without creating the dialog box.

`dStorage` A pointer to the memory for the dialog record. If you set this parameter to `NIL` for modal dialog boxes and movable modal dialog boxes, the Dialog Manager automatically allocates memory for them in your application heap. For a modeless dialog box, however, you should allocate your own memory as you would for a window—otherwise, your heap could become fragmented.

`behind` A pointer to the window behind which the dialog box is to be placed on the desktop. Always set this parameter to the window pointer `Pointer(-1)` to bring the dialog box in front of all other windows.

DESCRIPTION

The `GetNewDialog` function creates a dialog record from the information in the dialog resource and returns a pointer to it. You can use this pointer with Window Manager or QuickDraw routines to manipulate the dialog box. If the dialog resource specifies that the dialog box should be visible, the dialog box is displayed. If the dialog resource specifies that the dialog box should initially be invisible, use the Window Manager procedure `ShowWindow` to display the dialog box.

If you supply a dialog color table resource with the same resource ID as the dialog resource, `GetNewDialog` uses the `NewColorDialog` function and returns a pointer to a color graphics port. If no dialog color table resource is present, `GetNewDialog` uses `NewDialog` to return a pointer to a black-and-white graphics port, although system software draws the window frame using the system's default colors.

The `dStorage` and `behind` parameters of `GetNewDialog` have the same meaning as they do in the Window Manager function `GetNewWindow`. Always set the `behind` parameter to `Pointer(-1)` to bring the dialog box to the front.

The dialog resource contains the resource ID of the dialog box's item list resource. After calling the Resource Manager to read the item list resource into memory (if it's not already in memory), `GetNewDialog` makes a copy of the item list resource and uses that copy; thus you may have several dialog boxes with identical items.

If you provide a dialog color table resource, `GetNewDialog` copies it before passing it to the Window Manager routine `SetWinColor` unless the number-of-entries element of the dialog color table resource is set to –1, in which case the default window colors are used instead. The `GetNewDialog` function makes the copy so that the dialog color table resource can be purged without affecting the dialog box.

SPECIAL CONSIDERATIONS

The `GetNewDialog` function doesn't release the memory occupied by the resources. Therefore, your application should mark all resources used for a dialog box as purgeable.

If either the dialog resource or the item list resource can't be read, the function result is `NIL`; your application should test to ensure that `NIL` is not returned before performing any more operations with the dialog box or its items.

For modal dialog boxes, the Dialog Manager function `ModalDialog` traps all events. This prevents your event loop from receiving activate events for your windows. Thus, if one of your application's windows is active when you use `GetNewDialog` to create a modal dialog box, you must explicitly deactivate that window before displaying the modal dialog box.

If you ever need to display a dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you do not use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to the dialog box that your application presents.

The `GetNewDialog` function uses either `NewDialog` or `NewColorDialog`, each of which generates an update event for the entire window contents. Thus, with the

exception of controls, items aren't drawn immediately. The Dialog Manager calls the Control Manager to draw controls, and the Control Manager draws them immediately. So the controls won't be drawn twice, the Dialog Manager calls the Window Manager procedure `ValidRect` for the enclosing rectangle of each control. If you find that there is too great a lag between the drawing of controls and the drawing of other items, try making the dialog box initially invisible and then calling the Window Manager procedure `ShowWindow` to show it.

SEE ALSO

See "Creating Dialog Boxes" beginning on page 6-23 and "Displaying Alert and Dialog Boxes" beginning on page 6-61 for discussions and examples of how to use `GetNewDialog`.

The `GetNewWindow` and `ShowWindow` procedures are described in the chapter "Window Manager" of this book. The Notification Manager is described in the chapter "Notification Manager" in *Inside Macintosh: Processes*.

"Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 discuss menu adjustment when your application displays dialog boxes. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about messages and control titles in dialog boxes.

NewColorDialog

To create a dialog box, you can use the `NewColorDialog` function, which returns a pointer to a color graphics port. Generally, you should instead use `GetNewDialog` to create a dialog box, because `GetNewDialog` takes information about the dialog box from a dialog 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 `NewColorDialog` function is also available as the `NewCDialog` function.

```
FUNCTION NewColorDialog (dStorage: Ptr; boundsRect: Rect;
                         title: Str255; visible: Boolean;
                         procID: Integer; behind: WindowPtr;
                         goAwayFlag: Boolean; refCon: LongInt;
                         items: Handle): CDialogPtr;
```

`dStorage` A pointer to the memory for the dialog record. If you set this parameter to `NIL` for modal dialog boxes and movable modal dialog boxes, the Dialog Manager allocates memory for them on your application heap. For a modeless dialog box, however, you should allocate your own memory as you would for a window—otherwise, your heap could become fragmented.

`boundsRect`
: A rectangle, given in global coordinates, that determines the size and position of the dialog box; these coordinates specify the upper-left and lower-right corners of the dialog box.

`title`
: A text string used for the title of a modeless or movable modal dialog box. You can specify an empty string (not `NIL`) for a title bar that contains no text.

`visible`
: A flag that specifies whether the dialog box should be drawn on the screen immediately. If you set this parameter to `FALSE`, the dialog box is not drawn until your application uses the Window Manager procedure `ShowWindow` to display it.

`procID`
: The window definition ID for the type of dialog box. Use the `dBoxProc` constant to specify modal dialog boxes, the `noGrowDocProc` constant to specify modeless dialog boxes, and the `movableDBoxProc` constant to specify movable modal dialog boxes.

`behind`
: A pointer to the window behind which the dialog box is to be placed on the desktop. Always set this parameter to the window pointer `Pointer(-1)` to bring the dialog box in front of all other windows.

`goAwayFlag`
: A flag to specify whether a modeless dialog box should have a close box in its title bar when the dialog box is active. If you set this parameter to `TRUE`, the dialog window has a close box in its title bar when the window is active; only modeless dialog boxes should have close boxes.

`refCon`
: A value that the Dialog Manager uses to set the `refCon` field of the dialog box's window record. Your application may store any value here for any purpose. 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. 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.

`items`
: A handle to an item list resource for the dialog box. You can get the handle by calling the Resource Manager function `GetResource` to read the item list resource into memory. Use the Memory Manager procedure `HNoPurge` to make the handle unpurgeable while you use it or use the Operating System utility function `HandToHand` to make a copy of the handle and use the copy.

DESCRIPTION

The `NewColorDialog` function creates a dialog box as specified by its parameters and returns a pointer to a color graphics port for the new dialog box. The first eight parameters (`dStorage` through `refCon`) are passed to the Window Manager function `NewCWindow`, which creates the dialog box. You can use this pointer with Window Manager or QuickDraw routines to manipulate the dialog box.

The Dialog Manager uses the default window colors for the dialog box. By using the system's default colors, you ensure that your application's interface is consistent with

that of the Finder and other applications. However, if you absolutely feel compelled to break from this consistency, you can use the Window Manager procedure `SetWinColor` to use your own dialog color table resource that specifies colors other than the default colors. Be aware, however, that nonstandard colors in your alert and dialog boxes may initially confuse your users.

The Window Manager creates an auxiliary window record for the color dialog box. You can access this record with the Window Manager function `GetAuxWin`. (The `dialogCItemhandle` field of the auxiliary window record points to the dialog box's item color table resource.) If the dialog box's content color isn't white, it's a good idea to call `NewColorDialog` with the `visible` flag set to `FALSE`. After the color table and color item list resource are installed, use the Window Manager procedure `ShowWindow` to display the dialog box if it's the frontmost window. If the dialog box is a modeless dialog box that is not in front, use the Window Manager procedure `ShowHide` to display it.

When specifying the size and position of the dialog box in the `boundsRect` parameter, you should generally try to center dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is more appropriate. Also ensure that the tops of dialog boxes (including the title bars of modeless and movable modal dialog boxes) lie below the menu bar when you position them on the main screen. You can use the Menu Manager function `GetMBarHeight` to determine the height of the menu bar.

SPECIAL CONSIDERATIONS

For modal dialog boxes, the Dialog Manager function `ModalDialog` traps all events. This prevents your event loop from receiving activate events for your windows. Thus, if one of your application's windows is active when you use `NewColorDialog` to create a modal dialog box, you must explicitly deactivate that window before displaying the modal dialog box.

If you ever need to display a dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you do not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to the dialog box that your application presents.

The `NewColorDialog` function generates an update event for the entire window contents. Thus, with the exception of controls, items aren't drawn immediately. The Dialog Manager calls the Control Manager to draw controls, and the Control Manager draws them immediately. So that the controls won't be drawn twice, the Dialog Manager

calls the Window Manager procedure `ValidRect` for the enclosing rectangle of each control. If you find that there is too great a lag between the drawing of controls and the drawing of other items, try making the dialog box initially invisible and then calling the Window Manager procedure `ShowWindow` to show it.

SEE ALSO

Window Manager routines are described in the chapter "Window Manager" in this book. The Notification Manager is described in the chapter "Notification Manager" in *Inside Macintosh: Processes*. See *Inside Macintosh: Memory* for a description of `HNoPurge`. See *Inside Macintosh: Operating System Utilities* for a description of `HandToHand`.

"Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 discuss menu bar adjustment when your application displays dialog boxes. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about messages and control titles in dialog boxes. The `GetResource` function is described in the chapter "Resource Manager" of *Inside Macintosh: More Macintosh Toolbox*.

NewDialog

To create a dialog box, you can use the `NewDialog` function, which returns a pointer to a black-and-white graphics port (although system software draws the window frame of the dialog box using the system's default window colors). Generally, you should instead use `GetNewDialog` to create a dialog box; `GetNewDialog` takes information about the dialog from a dialog 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 `NewDialog` function is identical to the `NewColorDialog` function, except that `NewDialog` returns a pointer to a black-and-white graphics port. See the discussion of `NewColorDialog` on page 6-115 for descriptions of the parameters that you also pass to `NewDialog`.

```
FUNCTION NewDialog (dStorage: Ptr; boundsRect: Rect;
                    title: Str255; visible: Boolean;
                    procID: Integer; behind: WindowPtr;
                    goAwayFlag: Boolean; refCon: LongInt;
                    items: Handle):  DialogPtr;
```

DESCRIPTION

The `NewDialog` function creates a dialog box as specified by its parameters and returns a pointer to a black-and-white graphics port for the new dialog box. The first eight parameters (`dStorage` through `refCon`) are passed to the Window Manager function `NewWindow`, which creates the dialog box.

When specifying the size and position of the dialog box in the `boundsRect` parameter, you should generally try to center dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is more appropriate. Also ensure that the tops of dialog boxes (including the title bars of modeless and movable modal dialog boxes) lie below the menu bar when you position them on the main screen. You can use the Menu Manager function `GetMBarHeight` to determine the height of the menu bar.

SEE ALSO

If you use a dialog color table resource to change the default window colors, use the `NewColorDialog` function, which returns a pointer to a color graphics port. See the description of `NewColorDialog` on page 6-115 for additional information common to both the `NewDialog` and `NewColorDialog` functions.

CloseDialog

To dismiss a dialog box for whose dialog record you allocated memory, use the `CloseDialog` procedure.

```
PROCEDURE CloseDialog (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `CloseDialog` procedure removes a dialog box from the screen and deletes it from the window list. The `CloseDialog` procedure releases the memory occupied by

- the data structures associated with the dialog box (such as its structure, content, and update regions)
- all the items in the dialog box (except for pictures and icons, which might be shared by other resources) and any data structures associated with them

Generally, you should provide memory for the dialog record of modeless dialog boxes when you create them. (You can let the Dialog Manager provide memory for modal and movable modal dialog boxes.) You should then use `CloseDialog` to close a modeless dialog box when the user clicks the close box or chooses Close from the File menu.

Because `CloseDialog` does not dispose of the dialog resource or the item list resource, it is important to make these resources purgeable. Unlike `GetNewDialog`, `NewColorDialog` does not use a copy of the item list resource. Thus, if you use `NewColorDialog` to create a dialog box, you may want to use `CloseDialog` to keep the item list resource in memory even if you didn't supply a pointer to the memory.

SEE ALSO

If you let the Dialog Manager allocate memory for the dialog box (by passing `NIL` in the `dStorage` parameter to the `GetNewDialog`, `NewColorDialog`, or `NewDialog` function), use the `DisposeDialog` procedure, described next, instead of `CloseDialog`.

DisposeDialog

To dismiss a dialog box for which the Dialog Manager supplies memory, use the `DisposeDialog` procedure. The `DisposeDialog` procedure is also available as the `DisposDialog` procedure.

```
PROCEDURE DisposeDialog (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DisposeDialog` procedure calls the `CloseDialog` procedure and, in addition, releases the memory occupied by the dialog box's item list resource and the dialog record. Call `DisposeDialog` when you're done with a dialog box if you pass `NIL` in the `dStorage` parameter to `GetNewDialog`, `NewColorDialog`, or `NewDialog`.

Generally, your application should not allocate memory for the dialog records of modal dialog boxes or movable modal dialog boxes. In these cases your application should use `DisposeDialog` when the user clicks the OK or Cancel button.

SEE ALSO

If you allocate memory for the dialog box (for example, by passing a pointer in the `dStorage` parameter to the `GetNewDialog`, `NewColorDialog`, or `NewDialog` function), use `CloseDialog`, described on page 6-119, instead of `DisposeDialog`.

Manipulating Items in Alert and Dialog Boxes

In many cases, you won't have to make any changes to alert or dialog boxes after you define them in the resource file. If you do need to make changes, use the Dialog Manager routines described in this section.

For most item manipulation, first call the `GetDialogItem` procedure to get the information about the item. You can then use other routines to manipulate that item. Use the `SetDialogItem` procedure if you use any of these other routines to change the item. You must also use `SetDialogItem` to install any of your own application-defined draw procedures. If you use `SetDialogItem`, make the dialog box initially invisible, change the item as appropriate, then make the dialog box visible by using the Window Manager procedure `ShowWindow`. (For information about manipulating text in an alert box or a dialog box, see "Handling Text in Alert and Dialog Boxes" beginning on page 6-129.)

You can dynamically add items to and remove items from a dialog box by using the AppendDITL and ShortenDITL procedures. These procedures are especially useful if you share a single item list resource among multiple dialog boxes, because you can then use AppendDITL or ShortenDITL to add or remove items as appropriate for individual dialog boxes. You typically make such dialog boxes invisible, use the AppendDITL and ShortenDITL procedures as appropriate, then make the dialog boxes visible by using the Window Manager procedure ShowWindow.

GetDialogItem

To get a handle to an item so that you can manipulate it (for example, to determine its current value, to change it, or to install a pointer to a draw procedure for an application-defined item), use the GetDialogItem procedure. The GetDialogItem procedure is also available as the GetDItem procedure.

```
PROCEDURE GetDialogItem (theDialog: DialogPtr; itemNo: Integer;
                         VAR itemType: Integer; VAR item: Handle;
                         VAR box: Rect);
```

theDialog A pointer to a dialog record.

itemNo A number corresponding to the position of an item in the dialog box's item list resource.

itemType A value that represents the type of item requested in the itemNo parameter. You can use any of these constants to determine the value returned in this parameter:

```
CONST
 ctrlItem    = 4;     {add this constant to the next }
                      { four constants}
 btnCtrl     = 0;     {standard button control}
 chkCtrl     = 1;     {standard checkbox control}
 radCtrl     = 2;     {standard radio button}
 resCtrl     = 3;     {control defined in a 'CNTL'}
 helpItem    = 1;     {help balloons}
 statText    = 8;     {static text}
 editText    = 16;    {editable text}
 iconItem    = 32;    {icon}
 picItem     = 64;    {QuickDraw picture}
 userItem    = 0;     {application-defined item}
 itemDisable = 128;   {add to any of the above to }
                      { disable it}
```

item	For an application-defined draw procedure, a pointer to the draw procedure (coerced to a handle), returned for the item specified in the itemNo parameter; for all other item types, a handle to the item.
box	The display rectangle (described in coordinates local to the dialog box), returned for the item specified in the itemNo parameter.

DESCRIPTION

The GetDialogItem procedure returns in its parameters the following information about the item numbered itemNo in the item list resource of the specified dialog box: in the itemType parameter, the item type; in the item parameter, a handle to the item (or, for application-defined draw procedures, the procedure pointer); and in the box parameter, the display rectangle for the item.

For most item manipulation, first use the GetDialogItem procedure to get the information about the item. You can then use other routines, such as GetDialogItemText and SetDialogItem, to determine and change the value of that item.

SEE ALSO

Listing 6-12 on page 6-49 illustrates the use of GetDialogItem in conjunction with GetDialogItemText to retrieve the text entered by a user in an editable text item. Listing 6-16 on page 6-58 illustrates the use of GetDialogItem in conjunction with SetDialogItem to install the draw procedure for an application-defined item into a dialog box. Listing 6-26 on page 6-83 illustrates the use of GetDialogItem to determine the current value of a checkbox in a dialog box.

SetDialogItem

After using the GetDialogItem procedure to get a handle to an item from a dialog box, use the SetDialogItem procedure to set or change the item. The SetDialogItem procedure is also available as the SetDItem procedure.

```
PROCEDURE SetDialogItem (theDialog: DialogPtr; itemNo: Integer;
                         itemType: Integer; item: Handle;
                         box: Rect);
```

theDialog	A pointer to a dialog record.
itemNo	A number corresponding to the position of an item in the dialog box's item list resource.
itemType	A value that represents the type of item in the itemNo parameter. To specify the value for this parameter, you can use any of the constants listed on page 6-121 for the itemType parameter of the GetDialogItem procedure.

item	For an application-defined item, a pointer to the draw procedure (coerced to a handle) for the item specified in the itemNo parameter; for all other item types, a handle to the item.
box	The display rectangle (described in coordinates local to the dialog box) for the item specified in the itemNo parameter.

DESCRIPTION

The SetDialogItem procedure sets the item specified by the itemNo parameter for the specified dialog box. This procedure installs the item without drawing it; typically you create an invisible dialog box, use SetDialogItem, then use the Window Manager procedure ShowWindow to draw the dialog box and its items.

SEE ALSO

Listing 6-16 on page 6-58 illustrates how to use SetDialogItem to install an application-defined draw procedure. The ShowWindow procedure is described in the chapter "Window Manager" of this book.

HideDialogItem

Although you should rarely need to do so, you can make an item in a dialog box invisible by using the HideDialogItem procedure. The HideDialogItem procedure is also available as the HideDItem procedure.

```
PROCEDURE HideDialogItem (theDialog: DialogPtr; itemNo: Integer);
```

theDialog	A pointer to a dialog record.
itemNo	A number corresponding to the position of an item in the dialog box's item list resource.

DESCRIPTION

The HideDialogItem procedure hides the item specified by itemNo by giving it a display rectangle that's off the screen. Specifically, if the left coordinate of the item's display rectangle is less than 8192 (hexadecimal $2000), HideDialogItem adds 16,384 (hexadecimal $4000) to both the left and right coordinates of the rectangle. If the item is already hidden (that is, if the left coordinate is greater than 8192), HideDialogItem does nothing. To redisplay an item that's been hidden by HideDialogItem, you can use the ShowDialogItem procedure.

SPECIAL CONSIDERATIONS

If your application needs to display a number of dialog boxes that are similar except for one or two items, it's generally easier to modify the common elements using the `AppendDITL` and `ShortenDITL` procedures than to use the `HideDialogItem` and `ShowDialogItem` procedures.

The rectangle for a static text item must always be at least as wide as the first character of the text.

You generally shouldn't use `HideDialogItem` to make an editable text item invisible, because as the user presses the Tab key, the Dialog Manager attempts to move the cursor to the hidden editable text item, where the user's subsequent keystrokes will be placed.

ShowDialogItem

To redisplay an item that has been hidden by the `HideDialogItem` procedure, use the `ShowDialogItem` procedure. The `ShowDialogItem` procedure is also available as the `ShowDItem` procedure.

```
PROCEDURE ShowDialogItem (theDialog: DialogPtr; itemNo: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an item in the dialog box's item list resource.

DESCRIPTION

The `ShowDialogItem` procedure redisplays the item specified in `itemNo` by restoring the display rectangle the item had prior to the `HideDialogItem` call. Specifically, if the left coordinate of the item's display rectangle is greater than 8192, `ShowDialogItem` subtracts 16,384 from both the left and right coordinates of the rectangle. If the item is already visible (that is, if the left coordinate is less than 8192), `ShowDialogItem` does nothing.

The `ShowDialogItem` procedure adds the rectangle that contained the item to the update region so that it will be drawn. Note that if the item is a control you define 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. If the item is an editable text item, `ShowDialogItem` activates it by calling the TextEdit procedure `TEActivate`.

FindDialogItem

To determine the item number of an item at a particular location in a dialog box, use the `FindDialogItem` function. The `FindDialogItem` function is also available as the `FindDItem` function.

```
FUNCTION FindDialogItem (theDialog: DialogPtr; thePt: Point)
                         : Integer;
```

`theDialog` A pointer to a dialog record.

`thePt` A point, specified in coordinates local to the dialog box.

DESCRIPTION

If the point specified in the parameter `thePt` lies within an item, `FindDialogItem` returns a number corresponding to the position of that item in the dialog box's item list resource. If the point doesn't lie within the item's rectangle, `FindDialogItem` returns –1. If items overlap, `FindDialogItem` returns the item number of the first item, in the item list resource, containing the point.

This function is useful for changing the cursor when it's over a particular item.

The `FindDialogItem` function returns 0 for the first item in the item list resource, 1 for the second, and so on. To get the proper item number before calling the `GetDialogItem` or `SetDialogItem` procedure, add 1 to `FindDialogItem`'s function result, as shown here:

```
theItem := FindDialogItem(theDialog, thePoint) + 1;
```

Note that `FindDialogItem` returns the item number of disabled items as well as enabled items.

AppendDITL

To add items to an existing dialog box while your application is running, use the `AppendDITL` procedure.

```
PROCEDURE AppendDITL (theDialog: DialogPtr; theDITL: Handle;
                      theMethod: DITLMethod);
```

`theDialog` A pointer to a dialog record. This is the dialog record to which you will add the item list resource specified in the parameter `theDITL`.

`theDITL` A handle to the item list resource whose items you want to append to the dialog box.

theMethod The manner in which you want the new items to be displayed in the existing dialog box. You can pass a negative value to offset the appended items from a particular item in the existing dialog box. You can also pass any of these constants:

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

DESCRIPTION

The AppendDITL procedure adds the items in the item list resource specified in the parameter theDITL to the items of a dialog box. This procedure is especially useful if several dialog boxes share a single item list resource, because you can use AppendDITL to add items that are appropriate for individual dialog boxes. Your application can use the Resource Manager function GetResource to get a handle to the item list resource whose items you wish to add.

In the parameter theMethod, you specify how to append the new items, as follows:

- If you use 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.
- If you use the appendDITLRight constant, AppendDITL appends the items to the right of the dialog box 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.
- If you use the appendDITLBottom constant, AppendDITL appends the items to the bottom of the dialog box 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.
- You can also append a list of items relative to an existing item by passing a negative number in the parameter theMethod. 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, the display rectangles of the appended items are offset relative to the upper-left corner of item number 2 in the dialog box.

You typically create an invisible dialog box, call the AppendDITL procedure, then make the dialog box visible by using the Window Manager procedure ShowWindow.

SPECIAL CONSIDERATIONS

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 should use the Resource Manager procedure ReleaseResource to release the memory occupied by the appended item list resource. Otherwise, if your application calls AppendDITL to add items to that dialog box again, the dialog box remains

modified by your previous call—for example, it will still be longer at the bottom if you previously used the `appendDITLBottom` constant.

The `AppendDITL` procedure is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `AppendDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `AppendDITL` is available.

SEE ALSO

Listing 6-13 on page 6-54 and Listing 6-14 on page 6-55 illustrate a typical use of `AppendDITL`. Figure 6-29 on page 6-52 shows the result of using the `overlayDITL` constant, Figure 6-30 on page 6-52 shows the result of using the `appendDITLRight` constant, Figure 6-31 on page 6-53 shows the result of using the `appendDITLBottom` constant, and Figure 6-32 on page 6-53 shows the result of using a negative number in the parameter `theMethod`.

The chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* describes the `GetResource` and `ReleaseResource` routines. The `Gestalt` function is described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities*. See the chapter "Window Manager" in this book for information about `ShowWindow`.

ShortenDITL

To remove items from an existing dialog box while your application is running, use the `ShortenDITL` procedure.

```
PROCEDURE ShortenDITL (theDialog: DialogPtr;
                       numberItems: Integer);
```

`theDialog` A pointer to a dialog record.

`numberItems`
The number of items to remove (starting from the last item in the item list resource).

DESCRIPTION

The `ShortenDITL` procedure removes the specified number of items from the dialog box. This procedure is especially useful if several dialog boxes share a single item list resource, because you can use `ShortenDITL` to remove items as necessary for individual dialog boxes.

You typically create an invisible dialog box, call the `ShortenDITL` procedure, then make the dialog box visible by using the Window Manager procedure `ShowWindow`. 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.

SPECIAL CONSIDERATIONS

The `ShortenDITL` procedure is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `ShortenDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `ShortenDITL` is available.

SEE ALSO

You can use the `CountDITL` function, described next, to determine the number of items in the dialog box's item list resource. See the chapter "Window Manager" in this book for information on the `ShowWindow` and `SizeWindow` procedures. The `Gestalt` function is described in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

CountDITL

You can determine the number of items in a dialog box by using the `CountDITL` function.

```
FUNCTION CountDITL (theDialog: DialogPtr): Integer;
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `CountDITL` function returns the number of current items in a dialog box. You typically use `CountDITL` in conjunction with `ShortenDITL` to remove items from a dialog box.

SPECIAL CONSIDERATIONS

The `CountDITL` function is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `CountDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `CountDITL` is available.

SEE ALSO

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

Handling Text in Alert and Dialog Boxes

The Dialog Manager provides several routines for manipulating text. You can use the `ParamText` procedure to supply text strings, such as document titles, dynamically in the static text items of alert and dialog boxes. The `GetDialogItemText` and `SetDialogItemText` procedures are useful for determining and changing text in both static text and editable text items. You can use the `SelectDialogItemText` procedure to select and highlight text in an editable text item.

When a dialog box containing an editable text item is active, use the `DialogCut` procedure to handle the Cut editing command, the `DialogCopy` procedure to handle the Copy command, the `DialogPaste` procedure to handle the Paste command, and the `DialogDelete` procedure to handle the Clear command.

Once you determine that an event occurs in a modeless or movable modal dialog box, you can use the `DialogSelect` function, which is described on page 6-139, to handle key-down events in editable text items automatically. The `ModalDialog` procedure uses `DialogSelect` to handle key-down events in the editable text items of modal dialog boxes.

ParamText

To substitute text strings in the static text items of your alert or dialog boxes while your application is running, use the `ParamText` procedure.

```
PROCEDURE ParamText (param0: Str255; param1: Str255;
                     param2: Str255; param3: Str255);
```

`param0` A text string to substitute for the special string `^0` in the static text items of all subsequently created alert and dialog boxes.

`param1` A text string to substitute for the special string `^1` in the static text items of all subsequently created alert and dialog boxes.

`param2` A text string to substitute for the special string `^2` in the static text items of all subsequently created alert and dialog boxes.

`param3` A text string to substitute for the special string `^3` in the static text items of all subsequently created alert and dialog boxes.

DESCRIPTION

The `ParamText` procedure replaces the special strings `^0` through `^3` in the static text items of all subsequently created alert and dialog boxes with the text strings you pass as parameters. Pass empty strings (not `NIL`) for parameters not used.

SPECIAL CONSIDERATIONS

The strings used in `ParamText` are stored in the low-memory global variable `DAStrings`, which specifies a set of string handles used by the Dialog Manager.

If the user launches a desk accessory in your application's partition and the desk accessory calls `ParamText`, it may change the text in your application's dialog box.

You should be very careful about using `ParamText` in modeless dialog boxes. If a modeless dialog box using `ParamText` is onscreen and you display another dialog box or alert box that also uses `ParamText`, both boxes will be affected by the latest call to `ParamText`.

The strings you pass in the parameters to `ParamText` cannot contain the special strings `^0` through `^3`, or else the procedure will enter an endless loop of substitutions in versions of system software earlier than 7.1.

Note that you should try to store text strings in resource files to facilitate translation into other languages; therefore, `ParamText` is best used for supplying text strings, such as document names, that the user specifies. To avoid problems with grammar and sentence structure when you localize your application, you should use `ParamText` to supply only one text string per screen message.

SEE ALSO

Listing 6-9 on page 6-47 and Listing 6-10 on page 6-48 show an example of how you can use `ParamText` to supply the title of the user's current document to your alert and dialog boxes. If you need to supply a default text string to an editable text item while your application is running, use `SetDialogItemText`. The `SetDialogItemText` procedure also allows you to set or change the entire text string for a static text item.

GetDialogItemText

After using the `GetDialogItem` procedure to get a handle to an editable text item or a static text item in a dialog box, you can use the `GetDialogItemText` procedure to get the text string contained in that item. The `GetDialogItemText` procedure is also available as the `GetIText` procedure.

```
PROCEDURE GetDialogItemText (item: Handle; VAR text: Str255);
```

`item`	A handle to an editable text item or a static text item in a dialog box.
`text`	The text contained within the item.

DESCRIPTION

The `GetDialogItemText` procedure returns, in the `text` parameter, the text of the given editable text or static text item.

SPECIAL CONSIDERATIONS

If the user types more than 255 characters in an editable text item, `GetDialogItemText` returns only the first 255.

SEE ALSO

Listing 6-12 on page 6-49 illustrates how to use `GetDialogItemText` to retrieve the text that a user types into an editable text item.

SetDialogItemText

After using the `GetDialogItem` procedure to get a handle to an editable text item or a static text item in a dialog box, you can use the `SetDialogItemText` procedure to display a particular text string in that item. The `SetDialogItemText` procedure is also available as the `SetIText` procedure.

```
PROCEDURE SetDialogItemText (item: Handle; text: Str255);
```

`item`	A handle to an editable text item or a static text item in a dialog box.
`text`	The text to display in the item.

DESCRIPTION

The `SetDialogItemText` procedure places the specified text in the specified item and draws the item. This procedure is useful for supplying a default text string—such as a document name—for an editable text item while your application is running.

SPECIAL CONSIDERATIONS

All strings should be stored in resource files to ease translation into other languages.

SEE ALSO

For static text items, the `ParamText` procedure, described on page 6-129, is useful when you need to determine and provide only a portion of a text string while your application is running.

SelectDialogItemText

To select and highlight text contained in an editable text item, use the `SelectDialogItemText` procedure. The `SelectDialogItemText` procedure is also available as the `SelIText` procedure.

```
PROCEDURE SelectDialogItemText (theDialog: DialogPtr;
                                itemNo: Integer;
                                strtSel: Integer;
                                endSel: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an editable text item in the dialog box's item list resource.
`strtSel`	A number representing the position of the first character to begin selecting.
`endSel`	A number representing one position past the last character to be selected.

DESCRIPTION

If the item in the `itemNo` parameter is an editable text item that contains text, the `SelectDialogItemText` procedure sets the text selection range to extend from the character position specified in the `strtSel` parameter up to but not including the character position specified in the `endSel` parameter. The selection range is highlighted unless `strtSel` equals `endSel`, in which case a blinking vertical bar is displayed to indicate an insertion point at that position. If the editable text item doesn't contain text, `SelectDialogItemText` displays the insertion point.

You can select the entire text by specifying the number 0 in the `strtSel` parameter and the number 32767 in the `endSel` parameter.

For example, if the user makes an unacceptable entry in the editable text item, your application can display an alert box reporting the problem and then use `SelectDialogItemText` to select the entire text so it can be replaced by a new entry. Without this procedure, the user would have to select the item before making the new entry.

SEE ALSO

For details about text selection range and character position, see the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogCut

When a dialog box containing an editable text item is active, use the `DialogCut` procedure to handle the Cut editing command. The `DialogCut` procedure is also available as the `DlgCut` procedure.

```
PROCEDURE DialogCut (theDialog: DialogPtr);
```

`theDialog`	A pointer to a dialog record.

DESCRIPTION

The `DialogCut` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TECut` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TECut` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogCopy

When a dialog box containing an editable text item is active, use the `DialogCopy` procedure to handle the Copy editing command. The `DialogCopy` procedure is also available as the `DlgCopy` procedure.

```
PROCEDURE DialogCopy (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogCopy` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TECopy` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TECopy` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogPaste

When a dialog box containing an editable text item is active, use the `DialogPaste` procedure to handle the Paste editing command. The `DialogPaste` procedure is also available as the `DlgPaste` procedure.

```
PROCEDURE DialogPaste (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogPaste` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TEPaste` to the selected editable text item. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TEPaste` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogDelete

When a dialog box containing an editable text item is active, use the `DialogDelete` procedure to handle the Clear editing command. The `DialogDelete` procedure is also available as the `DlgDelete` procedure.

```
PROCEDURE DialogDelete (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogDelete` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TEDelete` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TEDelete` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

Handling Events in 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 both alert and modal dialog boxes, you should also provide a simple event filter function that allows other windows to respond to update events and that allows your alert or dialog box to respond to a few key-down events for keys such as Return, Enter, and Esc.

You can use your normal event-handling code to determine whether an event occurs in a modeless or movable modal dialog box, or you can use the `IsDialogEvent` function to learn whether they need to be handled as part of a dialog box. Once you determine that an event occurs in a modeless or movable modal dialog box, you can use the `DialogSelect` function to handle key-down events in editable text items automatically, to handle update and activate events automatically, and to report the enabled items clicked by the user. You then respond as appropriate to clicks in your active items. Or you can use Control Manager, TextEdit, and Window Manager routines (such as `FindWindow`, `BeginUpdate`, `EndUpdate`, `FindControl`, `TrackControl`, and `TEClick`) to handle these events without the aid of the Dialog Manager.

ModalDialog

To handle events when you display a modal dialog box, use the `ModalDialog` procedure.

```
PROCEDURE ModalDialog (filterProc: ModalFilterProcPtr;
                       VAR itemHit: Integer);
```

`filterProc`
A pointer to an event filter function.

`itemHit` A number representing the position of the selected item in the item list resource for the active modal dialog box.

DESCRIPTION

Call the `ModalDialog` procedure immediately after displaying a modal dialog box. The `ModalDialog` procedure assumes that a modal dialog box is displayed as the current port, and `ModalDialog` repeatedly handles events inside that port until an event involving an enabled dialog box item—such as a click in a radio button, for example—occurs. If the event is a mouse-down event outside the content region of the dialog box, `ModalDialog` emits the system alert sound and gets the next event. After receiving an event involving an enabled item, `ModalDialog` returns its item number in the `itemHit` parameter. Your application should then do whatever is appropriate in response to an event in that item. Your application should continue calling `ModalDialog` until the user selects the OK or Cancel button.

For events inside the dialog box, ModalDialog passes the event to the event filter function pointed to in the filterProc parameter before handling the event. When the event filter returns FALSE, ModalDialog handles the event. If the event filter function handles the event, the event filter function returns TRUE, and ModalDialog performs no more event handling.

If you set the filterProc parameter to NIL, the standard event filter function is executed. The standard event filter function returns TRUE and causes ModalDialog to return item number 1, which is the number of the default button, when the user presses the Return key or the Enter key. However, your application should provide a simple event filter function that

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

You can use the same event filter function in most or all of your alert and modal dialog boxes.

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

To handle events, ModalDialog calls the IsDialogEvent function. If the result of IsDialogEvent is TRUE, then ModalDialog calls the DialogSelect function to handle the event. Unless the event filter function returns TRUE, ModalDialog handles the event as follows:

- In response to an activate or update event for the dialog box, ModalDialog activates or updates its window.
- If the user presses the mouse button while the cursor is in an editable text item, ModalDialog responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If a key-down event occurs and there's an editable text item, ModalDialog uses TextEdit to handle text entry and editing automatically. If the editable text item is enabled, ModalDialog returns its item number after it receives either the mouse-down or key-down event. Normally, editable text items are disabled, and you use the GetDialogItemText procedure to read the information in the items only after the user clicks the OK button.
- If the user presses the mouse button while the cursor is in a control, ModalDialog calls the Control Manager function TrackControl. If the user releases the mouse button while the cursor is in an enabled control, ModalDialog returns the control's item number. Your application should respond appropriately—for example, by performing a command after the user clicks the OK button.

- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `ModalDialog` returns the item's number, and your application should respond appropriately. Generally, only controls should be enabled. If your application creates a control more complex than a button, radio button, or checkbox, your application must handle events inside that item with your event filter function.
- If the user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `ModalDialog` does nothing.

SPECIAL CONSIDERATIONS

Do not use `ModalDialog` for movable modal dialog boxes (that is, those created with the `movableDBoxProc` window definition ID) or for modeless dialog boxes (that is, those created with the `noGrowDocProc` window definition ID). If you want the Dialog Manager to assist you in handling events for movable modal and modeless dialog boxes, use the `IsDialogEvent` and `DialogSelect` functions instead.

The `ModalDialog` procedure calls the Event Manager function `GetNextEvent` with a mask that excludes disk-inserted events. To receive disk-inserted events, your event filter function can call the Event Manager procedure `SetSystemEventMask`.

When `ModalDialog` calls `TrackControl`, it does not allow you to specify the action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control, a picture, or an application-defined item that draws a control- like object in your dialog box. You must then provide an event filter function that appropriately handles events in that item.

SEE ALSO

Listing 6-26 on page 6-83 illustrates the use of `ModalDialog`. "Responding to Events in Editable Text Items" beginning on page 6-79 describes how `ModalDialog` uses TextEdit to handle text entry and editing in editable text items. The `IsDialogEvent` and `DialogSelect` functions (which your application may use instead of `ModalDialog` for modeless and movable modal dialog boxes) are described on page 6-138 and page 6-139, respectively. See the description of `MyEventFilter` on page 6-145 for information about the event filter function your application should specify in the `filterProc` parameter.

The `GetNextEvent` and `SetSystemEventMask` routines are described in the chapter "Event Manager" in this book. See that chapter as well for a discussion of disk-inserted events. See "Responding to Events in Controls" on page 6-78 for a description of how your application should respond to events inside of controls; the `TrackControl` function is fully described in the chapter "Control Manager" in this book. Also see that chapter for information about creating your own nonstandard controls. TextEdit is described in the chapter "TextEdit" of *Inside Macintosh: Text*.

IsDialogEvent

To determine whether a modeless dialog box or a movable modal dialog box is active when an event occurs, you can use the `IsDialogEvent` function.

```
FUNCTION IsDialogEvent (theEvent: EventRecord): Boolean;
```

`theEvent` An event record returned by an Event Manager function such as `WaitNextEvent`.

DESCRIPTION

If any event, including a null event, occurs when your dialog box is active, `IsDialogEvent` returns `TRUE`; otherwise, it returns `FALSE`. When `IsDialogEvent` returns `FALSE`, pass the event to the rest of your event-handling code. When `IsDialogEvent` returns `TRUE`, pass the event to `DialogSelect` after testing for the events that `DialogSelect` does not handle.

A dialog record includes a window record. When you use the `GetNewDialog`, `NewDialog`, or `NewColorDialog` function to create a dialog box, the Dialog Manager sets the `windowKind` field in the window record to `dialogKind`. To determine whether the active window is a dialog box, `IsDialogEvent` checks the `windowKind` field.

Before passing the event to `DialogSelect`, you should perform the following tests whenever `IsDialogEvent` returns `TRUE`:

- Check whether the event is a key-down event for the Return, Enter, Esc, or Command-period keystrokes. When the user presses the Return or Enter key, your application should respond as if the user had clicked the default button; when the user presses Esc or Command-period, your application should respond as if the user had clicked the Cancel button. Use the Control Manager procedure `HiliteControl` to highlight the applicable button for 8 ticks.
- At this point, you may also want to check for and respond to any special events that you do not wish to pass to `DialogSelect` or that require special processing before you pass them to `DialogSelect`. You would need to do this, for example, if the dialog box needs to respond to disk-inserted events.
- Check whether the event is an update event for a window other than the dialog box and, if it is, update your window.
- For complex items that you create, such as pictures or application-defined items that emulate complex controls, test for and respond to mouse events inside those items as appropriate. When `DialogSelect` calls `TrackControl`, it does not allow you to specify the action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control or a picture or an application-defined item that draws a control-like object in your dialog box. You must then test for and respond to those events yourself.

If your application uses `IsDialogEvent` to help handle events when you display a movable modal dialog box, perform the following additional tests before passing events to `DialogSelect`:

- Test for mouse-down events in the title bar of the movable modal dialog box and respond by dragging the dialog box accordingly.
- Test for and respond to mouse-down events in the Apple menu and, if the movable modal dialog box includes editable text items, in the Edit menu. (You should disable all other menus when you display a movable modal dialog box.)
- Play the system alert sound for every other mouse-down event outside the movable modal dialog box.

SPECIAL CONSIDERATIONS

Both `IsDialogEvent` and `DialogSelect` are unreliable when running in versions of system software earlier than System 7. You shouldn't use these routines if you expect your application to run in earlier versions of system software.

SEE ALSO

The `WaitNextEvent` function is described in the chapter "Event Manager" in this book. See *Inside Macintosh: Sound* for a description of the `SysBeep` procedure. The `FrontWindow` function is described in the chapter "Window Manager" in this book.

DialogSelect

After determining that an event related to an active modeless dialog box or an active movable modal dialog box has occurred, you can use the `DialogSelect` function to handle most of the events inside the dialog box.

```
FUNCTION DialogSelect (theEvent: EventRecord;
                       VAR theDialog: DialogPtr;
                       VAR itemHit: Integer): Boolean;
```

`theEvent` An event record returned by an Event Manager function such as `WaitNextEvent`.

`theDialog` A pointer to a dialog record for the dialog box where the event occurred.

`itemHit` A number corresponding to the position of an item within the item list resource of the active dialog box.

DESCRIPTION

The `DialogSelect` function handles most of the events relating to a dialog box. If the event is an activate or update event for a dialog box, `DialogSelect` activates or updates it and returns `FALSE`. If the event involves an enabled item, `DialogSelect`

returns a function result of TRUE. In its itemHit parameter, it returns the item number of the item selected by the user. In the parameter theDialog, it returns a pointer to the dialog record for the dialog box where the event occurred. In all other cases, the DialogSelect function returns FALSE. When DialogSelect returns TRUE, do whatever is appropriate as a response to the event involving that item in that particular dialog box; when it returns FALSE, do nothing.

Generally, only controls should be enabled in a dialog box; therefore your application should normally respond only when DialogSelect returns TRUE after the user clicks an enabled control, such as the OK button.

The DialogSelect function first obtains a pointer to the window containing the event. For update and activate events, the event record contains the window pointer. For other types of events, DialogSelect calls the Window Manager function FrontWindow. The Dialog Manager then makes this window the current graphics port by calling the QuickDraw procedure SetPort. Then DialogSelect prepares to handle the event by setting up text information if there are any editable text items in the active dialog box.

If the event is an update event for a dialog box, DialogSelect calls the Window Manager procedure BeginUpdate, the Dialog Manager procedure DrawDialog, and then the Window Manager procedure EndUpdate. 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.

The DialogSelect function handles the event as follows:

- In response to an activate or update event for the dialog box, DialogSelect activates or updates its window and returns FALSE.
- If a key-down event or an auto-key event occurs and there's an editable text item in the dialog box, DialogSelect uses TextEdit to handle text entry and editing, and DialogSelect returns TRUE for a function result. In its itemHit parameter, DialogSelect returns the item number.
- If a key-down event or an auto-key event occurs and there's no editable text item in the dialog box, DialogSelect returns FALSE.
- If the user presses the mouse button while the cursor is in an editable text item, DialogSelect responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If the editable text item is disabled, DialogSelect returns FALSE. If the editable text item is enabled, DialogSelect returns TRUE and in its itemHit parameter returns the item number. Normally, editable text items are disabled, and you use the GetDialogItemText function to read the information in the items only after the OK button is clicked.
- If the user presses the mouse button while the cursor is in a control, DialogSelect calls the Control Manager function TrackControl. If the user releases the mouse button while the cursor is in an enabled control, DialogSelect returns TRUE for a function result and in its itemHit parameter returns the control's item number. Your application should respond appropriately—for example, by performing a command after the user clicks the OK button.

- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `DialogSelect` returns `TRUE` for a function result and in its `itemHit` parameter returns the item's number. Generally, only controls should be enabled. If your application creates a complex control—such as one that measures how far a dial is moved—your application must handle mouse events in that item before passing the event to `DialogSelect`.
- If the user presses the mouse button while the cursor is in a disabled item, or if it is in no item, or if any other event occurs, `DialogSelect` does nothing.
- If the event isn't one that `DialogSelect` specifically checks for (if it's a null event, for example), and if there's an editable text item in the dialog box, `DialogSelect` calls the TextEdit procedure `TEIdle` to make the insertion point blink.

SPECIAL CONSIDERATIONS

Because `DialogSelect` handles only mouse-down events in a dialog box and key-down events in a dialog box's editable text items, you should handle other events as appropriate before passing them to `DialogSelect`. Likewise, when `DialogSelect` calls `TrackControl`, it does not allow you to specify any action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control or a picture or an application-defined item that draws a control-like object in your dialog box. You must then test for and respond to those events yourself.

Within dialog boxes, use the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support Cut, Copy, Paste, and Clear commands in editable text boxes.

The `DialogSelect` function is unreliable when running in versions of system software earlier than System 7. You shouldn't use this routine if you expect your application to run under earlier versions of system software.

SEE ALSO

Listing 6-25 on page 6-79 illustrates the use of `DialogSelect` to make the cursor blink in editable text items during null events; Listing 6-29 on page 6-92 illustrates the use of `DialogSelect` to handle mouse events in a modeless dialog box; Listing 6-33 on page 6-96 illustrates the use of `DialogSelect` to handle key-down events in editable text items; Listing 6-34 on page 6-98 illustrates the use of `DialogSelect` to handle activate events in a modeless dialog box.

DrawDialog

If you don't use any other Dialog Manager routines for handling events in a dialog box, you can use the `DrawDialog` procedure to draw its entire contents.

```
PROCEDURE DrawDialog (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DrawDialog` procedure draws the entire contents of the specified dialog box. The `DrawDialog` procedure draws all dialog items, calls the Control Manager procedure `DrawControls` to draw all controls, and calls the TextEdit procedure `TEUpdate` to update all static and editable text items and to draw their display rectangles. The `DrawDialog` procedure also calls the application-defined items' draw procedures if the items' rectangles are within the update region.

The `DialogSelect`, `ModalDialog`, `Alert`, `StopAlert`, `NoteAlert`, and `CautionAlert` routines use `DrawDialog` automatically. If you use `GetNewDialog` to create a dialog box but don't use any of these other Dialog Manager routines when handling events in the dialog box, you can use `DrawDialog` to redraw the contents of the dialog box when it's visible. If the dialog box is invisible, first use the Window Manager procedure `ShowWindow` and then use `DrawDialog`.

SEE ALSO

See the chapters "Window Manager" and "Event Manager" in this book for more information on update and activate events for windows. The `DrawControls` procedure is described in the chapter "Control Manager" in this book. The `TEUpdate` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

UpdateDialog

You can use the `UpdateDialog` procedure to redraw the update region of a specified dialog box. The `UpdateDialog` procedure is also available as the `UpdtDialog` procedure.

```
PROCEDURE UpdateDialog (theDialog: DialogPtr;
                        updateRgn: RgnHandle);
```

`theDialog` A pointer to a dialog record.

`updateRgn` A handle to the window region that needs to be updated.

DESCRIPTION

The `UpdateDialog` procedure redraws only the region in a dialog box specified in the `updateRgn` parameter. Because the `DialogSelect`, `ModalDialog`, `Alert`, `StopAlert`, `NoteAlert`, and `CautionAlert` routines automatically call `DrawDialog` to handle update events in your alert and dialog boxes, your application might never need to use `UpdateDialog`.

Instead of drawing the entire contents of the specified dialog box, `UpdateDialog` draws only the items in the specified update region. You can use `UpdateDialog` in response to an update event, and you should usually bracket it by calls to the Window Manager procedures `BeginUpdate` and `EndUpdate`. The `UpdateDialog` procedure uses the QuickDraw procedure `SetPort` to make the dialog box the current graphics port. For drawing controls, `UpdateDialog` uses the Control Manager procedure `UpdateControls`, which is faster than the `DrawControls` procedure.

SEE ALSO

Listing 6-35 on page 6-99 illustrates the use of `UpdateDialog` to respond to update events in a modeless dialog box. See the chapter "Window Manager" in this book for more information on update and activate events for windows. The `UpdateControls` procedure is described in the chapter "Control Manager" in this book.

Application-Defined Routines

If you supply an application-defined item in a dialog box, you must provide a draw procedure for the Dialog Manager to use when displaying the item; that procedure is referred to in this section as `MyItem`. If you want the Dialog Manager to play sounds other than the system alert sound, you must provide your own sound procedure, referred to in this section as `MyAlertSound`. To supplement the Dialog Manager's ability to handle events in the Macintosh multitasking environment, you should provide an event filter function that the Dialog Manager calls whenever it displays alert boxes and modal dialog boxes. This function is referred to as `MyEventFilter`.

MyItem

To draw your own application-defined item in a dialog box, provide a draw procedure that takes two parameters: a window pointer to the dialog box and an item number from the dialog box's item list resource. For example, this is how you should declare the procedure if you were to name it `MyItem`:

```
PROCEDURE MyItem (theWindow: WindowPtr; itemNo: Integer);
```

`theWindow` A pointer to the dialog record for the dialog box containing an application-defined item. If your procedure can draw in more than one dialog box, this parameter tells your procedure which one to draw in.

itemNo A number corresponding to the position of an item in the item list resource for the specified dialog box. If your procedure draws more than one item, this parameter tells your procedure which one to draw.

DESCRIPTION

The Dialog Manager calls your procedure to draw an application-defined item at the time you display the specified dialog box. When calling your draw procedure, the Dialog Manager sets the current port to the dialog box's graphics port. Normally, you create an invisible dialog box and then use the Window Manager procedure `ShowWindow` to display the dialog box.

Before you display the dialog box, use the `SetDialogItem` procedure to install this procedure in the dialog record. Before using `SetDialogItem`, you must first use the `GetDialogItem` procedure to obtain a handle to an item of type `userItem`.

If you enable the application-defined item that you draw with this procedure, the `ModalDialog` procedure and the `DialogSelect` function return the item's number when the user clicks that item. If your application needs to respond to a user action more complex than this (for example, if your application needs to measure how long the user holds down the mouse or how far the user drags the cursor), your application must track the cursor itself. If you use `ModalDialog`, your event filter function must handle events inside the item; if you use `DialogSelect`, your application must handle events inside the item before handing events to `DialogSelect`.

SEE ALSO

Listing 6-17 on page 6-59 illustrates a procedure that draws a bold outline around a button of any size and shape; Listing 6-16 on page 6-58 shows the use of `GetDialogItem` and `SetDialogItem` to install this draw procedure in a dialog record. The `ShowWindow` procedure is described in the chapter "Window Manager" in this book.

MyAlertSound

If you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure and call the `ErrorSound` procedure to make it the current sound procedure. For example, you can declare a sound procedure named `MyAlertSound`, as shown here:

```
PROCEDURE MyAlertSound (soundNo: Integer);
```

soundNo An integer from 0 to 3, representing the four possible alert stages.

DESCRIPTION

For each of the four alert stages that can be reported in the `soundNo` parameter, your procedure can emit any sound that you define. When the Dialog Manager calls your procedure, it passes 0 as the sound number for alert sounds specified by the `silent` constant in the alert resource. The Dialog Manager passes 1 for sounds specified by the `sound1` constant, 2 for sounds specified by the `sound2` constant, and 3 for sounds specified by the `sound3` constant.

SPECIAL CONSIDERATIONS

When the Dialog Manager detects a click outside an alert box or a modal dialog box, it uses the Sound Manager procedure `SysBeep` to play the system alert sound. By changing settings in the Sound control panel, the user can select which sound to play as the system alert sound. For consistency with system software and other Macintosh applications, your sound procedure should call `SysBeep` whenever your sound procedure receives sound number 1 (which you can represent with the `sound1` constant).

SEE ALSO

Listing 6-3 on page 6-22 illustrates how to use `MyAlertSound`. The `SysBeep` procedure is described in *Inside Macintosh: Sound.*

MyEventFilter

To supplement the Dialog Manager's ability to handle events, your application should provide an event filter function that the Dialog Manager calls when it displays alert boxes and modal dialog boxes. Your event filter function should have three parameters and return a Boolean value. For example, this is how you would declare it if you were to name it `MyEventFilter`:

```
FUNCTION MyEventFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

`theDialog`	A pointer to a dialog record for an alert box or a modal dialog box.
`theEvent`	An event record returned by an Event Manager function such as `WaitNextEvent`.
`itemHit`	A number corresponding to the position of an item in the item list resource for the alert or modal dialog box.

DESCRIPTION

After receiving an event that it does not handle, your function should return FALSE. When your function returns FALSE, ModalDialog handles the event, which you pass in the parameter theEvent. (Your function can also change the event to simulate a different event and return FALSE, which passes the event to the Dialog Manager for handling.) If your function *does* handle the event, your function should return TRUE as a function result, and in the itemHit parameter return the number of the item that it handled. The ModalDialog procedure and, in turn, the Alert, NoteAlert, StopAlert, and CautionAlert functions then return this item number in their own itemHit parameters.

Your event filter function should perform the following tasks:

- return TRUE and the item number for the default button if the user presses Return or Enter
- return TRUE and the item number for the Cancel button if the user presses Esc or Command-period
- update your windows in response to update events (this 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 in an application-defined item. For example, if you provide an application-defined item that requires you to measure how long the user holds down the mouse button or how far the user drags the cursor, use the event filter function to handle events inside that item.

The ModalDialog procedure calls the Event Manager function GetNextEvent with a mask that excludes disk-inserted events; to receive disk-inserted events, your event filter function can call the Event Manager procedure SetSystemEventMask.

You can use the same event filter function in most or all of your alert and modal dialog boxes.

For alert and modal dialog boxes, the Dialog Manager provides a standard event filter function that checks whether the user has pressed the Enter or Return key and, if so, returns the item number of the default button. Your event filter function should always check whether the Return key or Enter key was pressed and, if so, return the number of the default button in the itemHit parameter and a function result of TRUE.

In all alert and dialog boxes, any buttons that are activated by key sequences should invert to indicate which item has been selected. Use the Control Manager procedure HiliteControl to invert a button for 8 ticks, long enough to be noticeable but not so long as to be annoying. The Control Manager performs this action whenever users click a button, and your application should do this whenever the user presses the keyboard equivalent of a button click.

For modal dialog boxes that contain editable text items, your application should handle menu bar access to allow use of your Edit menu and its Cut, Copy, Paste, Clear, and Undo commands. Your event filter function should then test for and handle clicks in your Edit menu and keyboard equivalents for the appropriate commands in your Edit menu. Your application should respond by using the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands.

For an alert box, you specify a pointer to your event filter function in a parameter that you pass to the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. For a modal dialog box, specify a pointer to your event filter function in a parameter that you pass to the `ModalDialog` procedure.

SEE ALSO

Listing 6-27 on page 6-88 illustrates an event filter function. The functions `GetNextEvent` and `SetSystemEventMask` are described in the chapter "Event Manager" in this book.

Resources

This section describes resources used by the Dialog Manager for displaying alerts and dialog boxes. These resources are

- the dialog (`'DLOG'`) resource, which specifies the window type, display rectangle, and item list resource for a dialog box
- the alert (`'ALRT'`) resource, which specifies alert sounds, a display rectangle, and an item list resource for an alert box
- the item list (`'DITL'`) resource, which specifies the items—such as buttons and static text—to display in an alert box or a dialog box
- the dialog color table (`'dctb'`) resource, which lets you supply a color graphics port for a dialog box and also use colors other than the default colors in a dialog box
- the alert color table (`'actb'`) resource, which lets you use colors other than the default colors in an alert box
- the item color table (`'ictb'`) resource, which lets you change the default colors, typeface, font style, and font size of items in an alert box or a dialog box

This section describes the structures of these resources after they are compiled by the Rez resource compiler, available from APDA. If you are interested in creating the Rez input files for these resources, see "Using the Dialog Manager" beginning on page 6-17 for detailed information.

The Dialog Resource

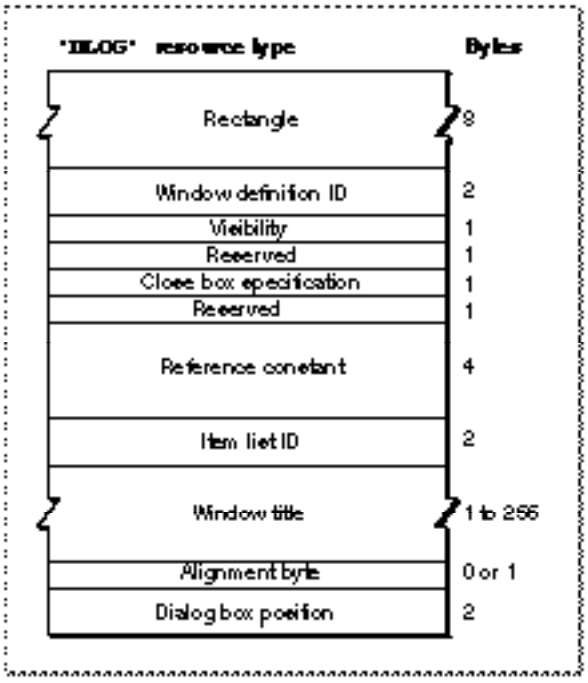
You can use a dialog resource to define a dialog box. A dialog resource is a resource of type `'DLOG'`. All dialog resources must be marked purgeable, and they must have resource ID numbers greater than 128.

To specify the items in a dialog box, you must also provide an item list resource, described beginning on page 6-151. Use the `GetNewDialog` function (described on page 6-113) to create the dialog box defined in the dialog resource.

The format of a Rez input file for a dialog resource differs from its compiled output format. This section describes the structure of a Rez-compiled dialog resource. If you are concerned only with creating a dialog resource, see "Creating Dialog Boxes" beginning on page 6-23.

Figure 6-42 shows the format of a compiled dialog resource.

Figure 6-42 Structure of a compiled dialog (`'DLOG'`) resource



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

- Rectangle. This determines the dialog box's dimensions and, possibly, its position. (The last element in the dialog resource usually specifies a position for the dialog box.)
- Window definition ID.
  - If the integer 0 appears here (as specified in the Rez input file by the `dBoxProc` window definition ID), the Dialog Manager displays a modal dialog box.

- If the integer 4 appears here (as specified in the Rez input file by the `noGrowDocProc` window definition ID), the Dialog Manager displays a modeless dialog box.
- If the integer 5 appears here (as specified in the Rez input file by the `movableDBoxProc` window definition ID), the Dialog Manager displays a movable modal dialog box.

These types of dialog boxes are illustrated in Figure 6-6 on page 6-10, Figure 6-8 on page 6-12, and Figure 6-7 on page 6-11, respectively.

- Visibility. If this is set to a value of 1 (as specified by the `visible` constant in the Rez input file), the Dialog Manager displays this dialog box as soon as you call the `GetNewDialog` function. If this is set to a value of 0 (as specified by the `invisible` constant in the Rez input file), the Dialog Manager does not display this dialog box until you call the Window Manager procedure `ShowWindow`.
- Close box specification. This specifies whether to draw a close box. Normally, this is set to a value of 1 (as specified by the `goAway` constant in the Rez input file) only for a modeless dialog box to specify a close box in its title bar. Otherwise, this is set to a value of 0 (as specified by the `noGoAway` constant in the Rez input file).
- Reference constant. This contains any value that an application stores here. For example, an 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. An application 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.
- Item list resource ID. The ID of the item list resource that specifies the items—such as buttons and static text—to display in the dialog box.
- Window title. This is a Pascal string displayed in the dialog box's title bar only when the dialog box is modeless.
- Alignment byte. This is an extra byte added if necessary to make the previous Pascal string end on a word boundary.
- Dialog box position. This specifies the position of the dialog box on the screen. (If your application positions dialog boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager.)
  - If 0x0000 appears here (as specified by the `noAutoCenter` constant in the Rez input file), the Dialog Manager positions this dialog box according to the global coordinates specified in the rectangle element of this resource.
  - If 0xB00A appears here (as specified by the `alertPositionParentWindow` constant in the Rez input file), the Dialog Manager positions the dialog box over the frontmost window so that the window's title bar appears. This is illustrated in Figure 6-33 on page 6-63.
  - If 0x300A appears here (as specified by the `alertPositionMainScreen` constant in the Rez input file), the Dialog Manager centers the dialog box near the top of the main screen. This is illustrated in Figure 6-34 on page 6-63.
  - If 0x700A appears here (as specified in the Rez input file by the `alertPositionParentWindowScreen` constant), the Dialog Manager positions the dialog box on the screen where the user is currently working. This is illustrated in Figure 6-35 on page 6-64.

The Alert Resource

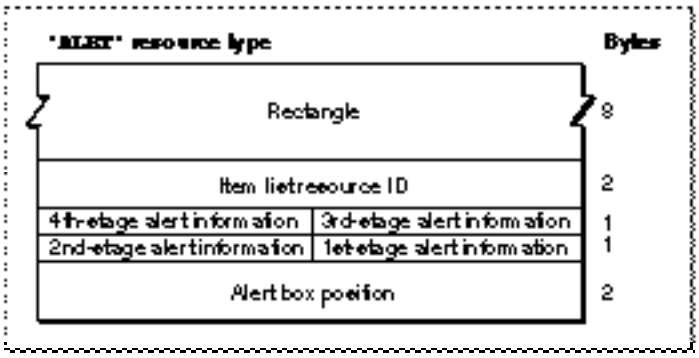
You can use an alert resource to define an alert. An alert resource is a resource of type `'ALRT'`. All alert resources must be marked purgeable, and they must have resource ID numbers greater than 128.

To specify the items in an alert box, you must also provide an item list resource, described beginning on page 6-151. To display the alert, you call either the `NoteAlert`, `CautionAlert`, `StopAlert`, or `Alert` function and pass it the resource ID of the alert resource. The `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert` functions are described in "Creating Alerts" beginning on page 6-105.

The format of a Rez input file for an alert resource differs from its compiled output format. This section describes the structure of a Rez-compiled alert resource. If you are concerned only with creating an alert resource, see "Creating Alert Sounds and Alert Boxes" beginning on page 6-18.

Figure 6-43 shows the structure of a compiled alert resource.

Figure 6-43 Structure of a compiled alert (`'ALRT'`) resource



The compiled version of an alert resource contains the following elements:

- Rectangle. This determines the alert box's dimensions and, possibly, its position. (The last element in the alert resource usually specifies a position for the alert box.)
- Item list resource ID. The ID of the item list resource that specifies the items—such as buttons and static text—to display in the alert box.
- Fourth-stage alert information. This specifies the response when the user repeats the action that invokes this alert four or more consecutive times. The Dialog Manager responds in the manner specified in the 4 bits that make up this element.
  - If the first bit is set, the Dialog Manager draws a bold outline around the second item in the item list resource (typically, the Cancel button) and—if your application does not specify an event filter function—returns 2 when the user presses the Return or Enter key at the fourth consecutive occurrence of the alert. If the first bit is not set, the Dialog Manager draws a bold outline around the first item in the item list resource (typically, the OK button) and—if your application does not specify an event filter function—returns 1 when the user presses the Return or Enter key.

- If the second bit is set, the Dialog Manager displays the alert box at this stage. If the second bit is not set, the Dialog Manager doesn't display the alert box at this stage.
- If neither of the next 2 bits is set, the Dialog Manager plays no alert sound at this stage. If bit 3 is set and bit 4 is not set, the Dialog Manager plays the first alert sound—by default, the system alert sound. If bit 3 is not set and bit 4 is set, the Dialog Manager plays the second alert sound; by default, it plays the system alert sound twice. If both bit 3 and bit 4 are set, the Dialog Manager plays the third alert sound; by default, it plays the system alert sound three times. By defining your own alert sound (described on page 6-144) and calling the `ErrorSound` procedure (described on page 6-104) to make it the current sound procedure, you can specify your own alert sounds.

- Third-stage alert information. This specifies the response when the user repeats the action that invokes this alert three consecutive times. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- Second-stage alert information. This specifies the response when the user repeats the action that invokes this alert two consecutive times. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- First-stage alert information. This specifies the response for the first time that the user performs the action that invokes this alert. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- Alert box position. This specifies the position of the alert box on the screen. (If your application positions alert boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager.)
  - If 0x0000 appears here (as specified by the `noAutoCenter` constant in the Rez input file), the Dialog Manager positions this alert box according to the global coordinates specified in the rectangle element of this resource.
  - If 0xB00A appears here (as specified by the `alertPositionParentWindow` constant in the Rez input file), the Dialog Manager positions the alert box over the frontmost window so that the window's title bar appears. This is illustrated in Figure 6-33 on page 6-63.
  - If 0x300A appears here (as specified by the `alertPositionMainScreen` constant in the Rez input file), the Dialog Manager centers the alert box near the top of the main screen. This is illustrated in Figure 6-34 on page 6-63.
  - If 0x700A appears here (as specified in the Rez input file by the `alertPositionParentWindowScreen` constant), the Dialog Manager positions the alert box on the screen where the user is currently working. This is illustrated in Figure 6-35 on page 6-64.

The Item List Resource

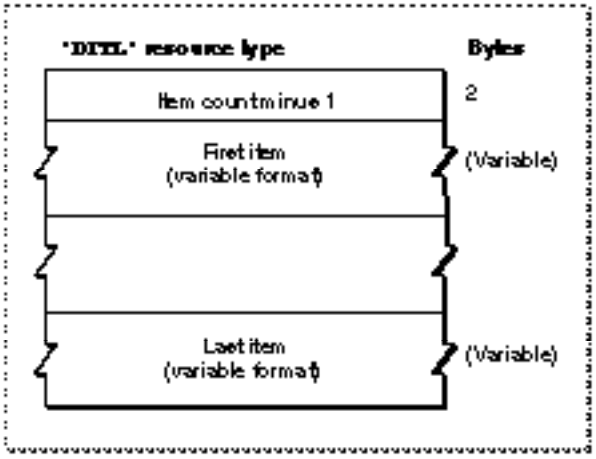
You use an item list resource to specify items—such as buttons and text—in alert boxes and dialog boxes. An item list resource is a resource with the resource type `'DITL'`. All item list resources must be marked purgeable, and they must have resource ID numbers greater than 128.

For an alert box, you specify the resource ID of the item list resource in an alert resource (described beginning on page 6-150). For a dialog box that you create with the `GetNewDialog` function, you specify the resource ID of the item list resource in a dialog resource (described beginning on page 6-148). For a dialog box that you create with either the `NewColorDialog` function (described on page 6-115) or the `NewDialog` function (described on page 6-118), you use the Resource Manager function `GetResource` to read the item list resource into memory and to provide a handle to the item list resource in memory.

The format of a Rez input file for an item list resource differs from its compiled output format. This section describes the structure of a Rez-compiled item list resource. If you are concerned only with creating an item list resource, see "Providing Items for Alert and Dialog Boxes" beginning on page 6-26.

Figure 6-44 shows the format of a compiled item list resource.

Figure 6-44 Structure of a compiled item list (`'DITL'`) resource



The compiled version of an item list resource contains the following elements:

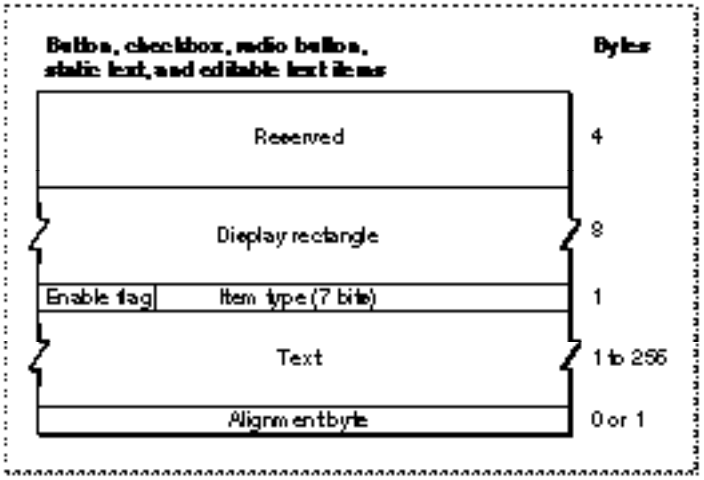
- Item count minus 1. This value is 1 less than the total number of items defined in this resource.
- A variable number of items.

The format of each item depends on its type. Figure 6-45 shows the format of an item defined to be a button, a checkbox, a radio button, a static text item, or an editable text item.

The compiled version of a button, checkbox, radio button, static text item, or editable text item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert box or dialog box; these coordinates specify the upper-left and lower-right corners of the item.

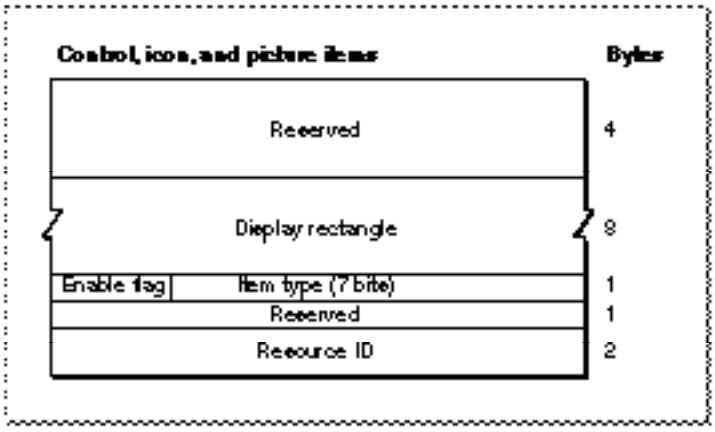
Figure 6-45 Structure of compiled button, checkbox, radio button, static text, and editable text items



- Enable flag. This specifies whether the item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type.
  - If this bit string is set to 4 (as specified in the Rez input file by the `Button` constant), then the item is a button.
  - If this bit string is set to 5 (as specified in the Rez input file by the `CheckBox` constant), then the item is a checkbox.
  - If this bit string is set to 6 (as specified in the Rez input file by the `RadioButton` constant), then the item is a radio button.
  - If this bit string is set to 8 (as specified in the Rez input file by the `StaticText` constant), then the item is static text.
  - If this bit string is set to 16 (as specified in the Rez input file by the `EditText` constant), then the item is editable text.
- Text. This specifies the text that appears in the item. This element consists of a length byte and as many as 255 additional bytes for the text. (“Titles for Buttons, Checkboxes, and Radio Buttons” beginning on page 6-37 and “Text Strings for Static Text and Editable Text Items” beginning on page 6-40 contain recommendations about appropriate text in items.)
  - For a button, checkbox, or radio button, this is the title for that control.
  - For a static text item, this is the text of the item.
  - For an editable text item, this can be an empty string (in which case the editable text item contains no text), or it can be a string that appears as the default string in the editable text item.
- Alignment byte. This is added if necessary to make the previous text string end on a word boundary.

Figure 6-46 shows the format for an element defined to be a control, an icon, or a picture item.

Figure 6-46 Structure of compiled control, icon, and picture items

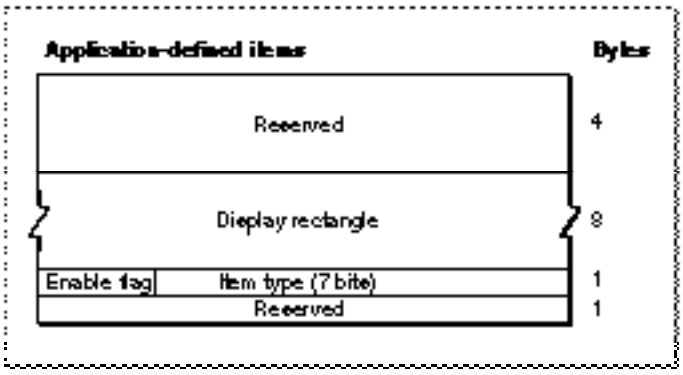


The compiled version of a control, an icon, or a picture item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert or dialog box.
- Enable flag. This specifies whether the item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type.
  - If this 7-bit string is set to 7 (as specified in the Rez input file by the `Control` constant), then the item is a button.
  - If this is set to 32 (as specified in the Rez input file by the `Icon` constant), then the item is an icon.
  - If this is set to 64 (as specified in the Rez input file by the `Picture` constant), then the item is a QuickDraw picture.
- Resource ID.
  - For a control item, this is the resource ID of a `'CTRL'` resource.
  - For an icon item, this is the resource ID of an `'ICON'` resource and, optionally, a `'cicn'` resource
  - For a picture item, this is the resource ID of a `'PICT'` resource.

Figure 6-47 shows the format for an application-defined item.

Figure 6-47 Structure of a compiled application-defined item

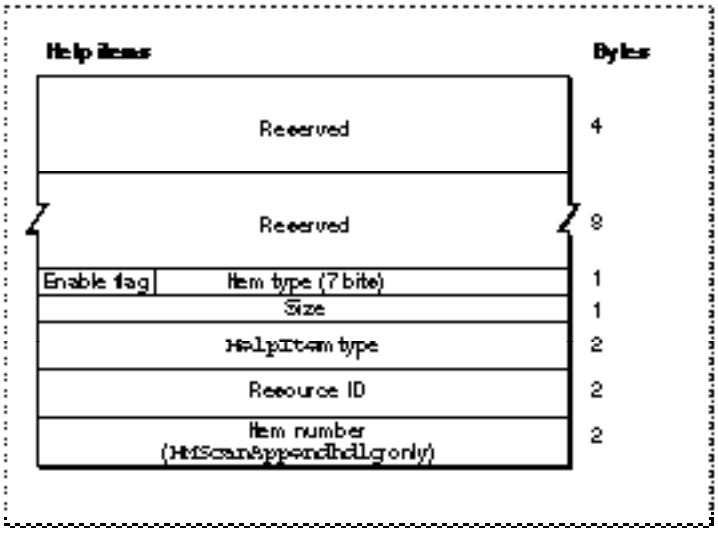


The compiled version of an application-defined item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the application-defined item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert box or dialog box.
- Enable flag. This specifies whether the application-defined item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type. This is set to a value of 0 (as specified in the Rez input file by the `UserItem` constant).

Figure 6-48 shows the format for a help item. (Help items are described in detail in the chapter "Help Manager" of *Inside Macintosh: More Macintosh Toolbox*.)

Figure 6-48 Structure of compiled help items



The compiled version of a help item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Reserved. This should be set to 0.
- Enable flag. This specifies whether the item is enabled or disabled. For help items, this bit should never be set, because the Dialog Manager cannot report to your application when mouse-down events occur inside the item.
- Item type. This is set to 1 (as specified in the Rez input file by the `HelpItem` constant).
- Size. This specifies the number of bytes contained in the rest of this element. This is set to 4 for an item identified by either the `HMScanhdlg` or `HMScanhrct` identifier, or it's set to 6 for an item identified by the `HMScanAppendhdlg` identifier.
- `HelpItem` type. This specifies the type of help item defined in the resource.
  - For an item identified by the `HMScanhdlg` identifier, this element contains the value 1.
  - For an item identified by the `HMScanhrct` identifier, this element contains the value 2.
  - For an item identified by the `HMScanAppendhdlg` identifier, this element contains the value 8.
- Resource ID. This is the resource ID of the resource containing the help messages for this alert box or dialog box.
  - For an item identified by either the `HMScanhdlg` or `HMScanAppendhdlg` identifier, this is the ID of an `'hdlg'` resource.
  - For an item identified by the `HMScanhrct` identifier, this is the ID of an `'hrct'` resource.
- Item number. This is available only for an item identified by the `HMScanAppendhdlg` identifier. This is the item number within the alert box or dialog box after which the help messages specified in the `'hdlg'` resource should be displayed. These help messages relate to the items that are appended to the alert box or dialog box. (The item list resource does not contain these 2 bytes for items identified by either the `HMScanhdlg` or `HMScanhrct` identifier.)

The Dialog Color Table Resource

On color monitors, the Dialog Manager automatically adds color to your alert and dialog boxes so that they match the colors of the windows, alert boxes, and dialog boxes used by system software. These colors provide aesthetic consistency across all monitors, from black-and-white displays to 8-bit color displays. On a color monitor, for example, the racing stripes in the title bar of a modeless dialog box are gray, the close box and window frame are in color, and the buttons and text are black.

When you create dialog resources, your application's dialog boxes use the system's default colors. Typically, this is all you need to do to provide color for your dialog boxes—with the following exceptions:

- When you need to include a color version of an icon in a dialog box, you must create a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource. Plate 2 at the front of this book shows an alert box that includes a color icon.

- When you need to produce a blended gray color for outlining the inactive (that is, dimmed) default button, you must create a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource.

"Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 explains how to create a draw routine that outlines the default button of a dialog box. If you deactivate a dialog box, you should dim its buttons and use gray to draw the outline for the default button. Because `GetNewDialog` and `NewDialog` supply black-and-white graphics ports for dialog boxes, you can create a dialog color table resource for the dialog box to force the Dialog Manager to supply a color graphics port. Then you can use a blended gray color for the outline for the default button. (The `NewColorDialog` function supplies a color graphics port.)

Even when you create a dialog color table resource for drawing a gray outline, you should not change the system's default colors. If you feel absolutely compelled to use nonstandard colors, you can use the Dialog Manager to specify colors other than the default colors. Your application can specify its own colors for a dialog box by creating a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource (described beginning on page 6-148). You don't have to call any new routines to change the colors used in dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load a dialog color table resource with the same resource ID as the dialog resource.

Be aware, however, that nonstandard colors in your dialog boxes may initially confuse your users. Also be aware that despite any changes you may make, users can alter the colors of dialog boxes anyway by changing settings in the Color control panel.

▲ **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. ▲

A dialog color table resource has exactly the same format as a window color table (that is, a resource of type `'wctb'`), which is described in the chapter "Window Manager" of this book.

If the dialog box's content color isn't white, specify the `invisible` constant in the dialog resource. Use the Window Manager procedure `ShowWindow` to display the dialog box when it's the frontmost window. If the dialog box is a modeless dialog box that is not in front, use the Window Manager procedure `ShowHide` to display it.

The Alert Color Table Resource

On color monitors, the Dialog Manager automatically adds color to your alert boxes so that they match the colors of the windows and alerts used by system software. When you create alert resources, your application's alert boxes use the system's default colors. Typically, this is all you need to do to provide color for your alert boxes. (However, to include a color version of an icon in an alert box, you must add a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource.)

If you feel absolutely compelled to use nonstandard colors, you can use the Dialog Manager to specify colors other than the default colors. Your application can specify its own colors for an alert box by creating an alert color table (`'actb'`) resource with the same resource ID as the alert resource (described beginning on page 6-150). You don't have to call any new routines to change the colors used in alert or dialog boxes. When you call the `Alert` function, for example, the Dialog Manager automatically attempts to load an alert color table resource with the same resource ID as the alert resource.

Be aware, however, that nonstandard colors in your alert boxes may initially confuse your users. Also be aware that despite any changes you may make, users can alter the colors of dialog boxes anyway by changing settings in the Color control panel.

▲ **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. ▲

An alert color table resource has exactly the same format as a window color table (`'wctb'`) resource, which is described in the chapter "Window Manager" of this book.

The Item Color Table Resource

On color monitors, the Dialog Manager automatically draws the items in your dialog and alert boxes so that they match the colors of the items used by system software in its dialog and alert boxes. The Dialog Manager also uses the default system font when it draws the text in the static text and editable text items of your dialog and alert boxes.

If you feel absolutely compelled to use nonstandard fonts and colors, you can use the Dialog Manager to specify your own colors, typeface, font style, and font size.

Note
The Dialog Manager displays the typeface, font style, and font size you specify only on color monitors. ◆

Your application can specify these by creating an item color table (`'ictb'`) resource with the same resource ID as the dialog or alert box's item list resource , and then providing a dialog color table resource for a dialog box or an alert color table resource for an alert box. You don't have to call any new routines to change the colors, typefaces, font styles, or font sizes used in dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load an item color table resource with the same resource ID as the item list resource.

Note
To make it easier to localize your application for other script systems, 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. ◆

Also, be aware that nonstandard colors for items in your dialog and alert boxes may initially confuse your users.

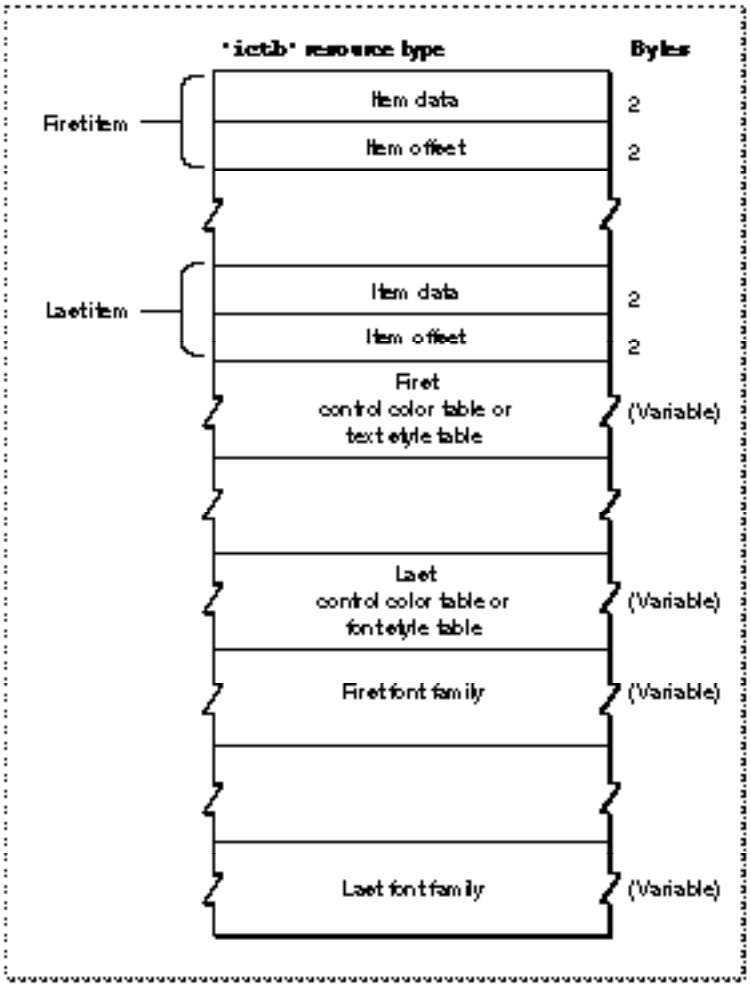
▲ **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. ▲

If you want to provide an item color table resource for an alert box or a dialog box, you must create an alert color table resource or a dialog color table resource, even if the item color table resource has no actual color information and describes only static text and editable text style changes.

An item color table resource is a resource of type `'ictb'`. All item color table resources must have resource ID numbers greater than 128.

There is no Rez template available for creating item color table resources. When you compile an item color table resource, it should follow the format illustrated in Figure 6-49.

Figure 6-49 Structure of a compiled item color table resource



You define an item color table resource for a dialog box or an alert box by specifying these elements in a resource with the `'ictb'` resource type:

- Items. These consist of a variable number of items, corresponding to those in an item list resource with the same resource ID as this item color table resource.
- Control color tables and text style tables.
  - A **control color table** defines the colors used in a control. Several controls can share the same control color table.
  - A **text style table** defines the font family, font style, font size, and color of text in an editable text item or a static text item. Several editable text and static text items can share the same text style table.
- Optionally, a list of font families. If you use any text style tables, you generally conclude the item color table resource with a list of text strings, each of which specifies a font family. Although you may specify font numbers instead of font names, it's much more reliable to specify names, because system software may renumber these fonts as they are installed and removed. For every editable text item and static text item listed at the top of the item color table resource, specify a font family at the bottom of the resource.

The information contained in an element depends on the type of item it describes:

- Item data. This contains information about how this item is described in the rest of this resource.
  - For a control, this is the length (in bytes) of its control color table.
  - For a static text item or an editable text item, the bits of this element determine which elements of the text style table to use and are interpreted as follows:

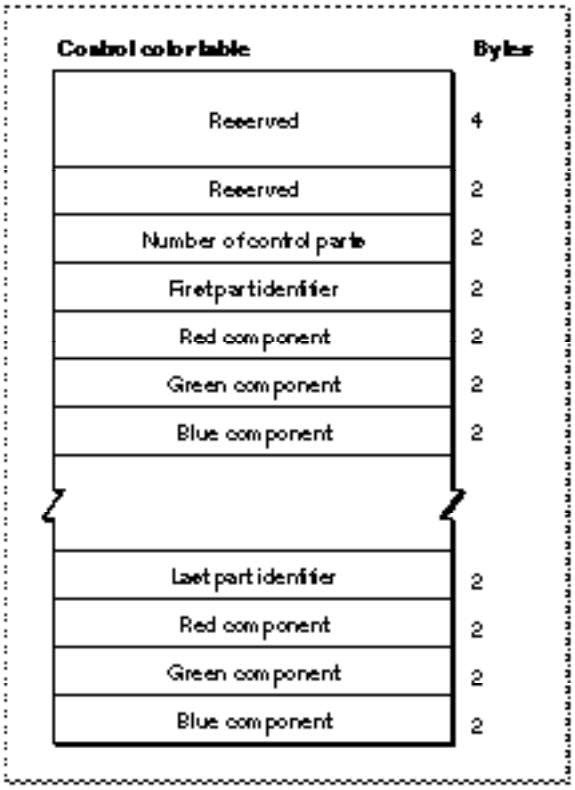
Bit	Meaning
0	Change the font family.
1	Change the typeface.
2	Change the font size.
3	Change the font foreground color.
4	Add the font size.
13	Change the font background color.
14	Change the font mode.
15	The font element is an offset to the name.

- Item offset. The number of bytes from the beginning of the resource to either the control color table or the text style table that describes this item.

When both the item data and item offset elements are set to 0, then the control or text item is drawn with the default colors, typeface, font size, and font style. Even if only the first few items of the dialog box have color style information, there must be room for all of the items actually in the box (with the item data and item offset elements of the unused entries set to 0).

For controls, the colors are described by a color table identical to a `'cctb'` resource used by the Control Manager. Multiple controls can use the same color table. If the resource sets both the item data and the item offset element to 0, then the system's default colors are used for the control. The format of a control color table is illustrated in Figure 6-50.

Figure 6-50 Structure of a compiled control color table



A control color table consists of the following elements:

- Reserved. This should always be set to a value of 0.
- Reserved. Again, should always be set to a value of 0.
- Number of control parts. For standard controls other than scroll bars, this should be set to 3, because a standard control uses only three parts: frame, control body, and text. For scroll bars, this should be set to 12; see the description of the control color table resource in the chapter "Control Manager" for information on specifying the colors for a scroll bar. To create a control that uses other parts, you must create a custom `'CDEF'` resource, as described in the chapter "Control Manager" in this book.

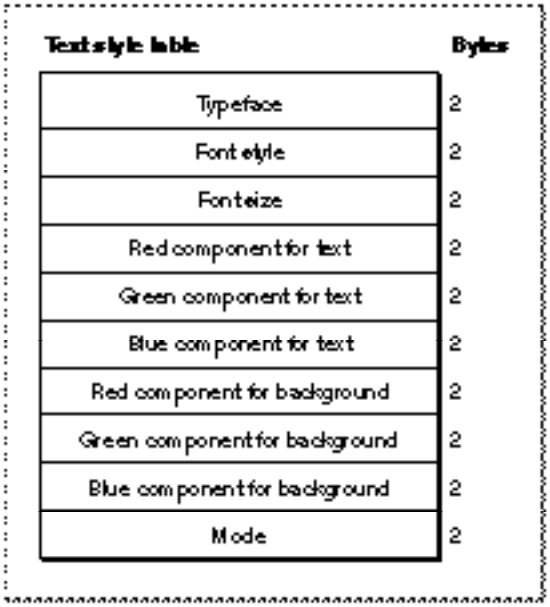
- Part identifier. This is a value that identifies a part of the first control. The following list shows the values and constants they represent for the standard controls other than scroll bars. For information on the part identifiers for a scroll bar, see the description of the control color table resource in the chapter "The Control Manager" in this book. They can be listed in any order in the control color table.

Constant	Value	Control part
`cFrameColor`	0	Frame
`cBodyColor`	1	Body
`cTextColor`	2	Text (such as titles)

- Red component. This is an integer that represents the intensity of the red component of the color to use when drawing this control part.
- Green component. This is an integer that represents the intensity of the green component of the color to use when drawing this control part.
- Blue component. This is an integer that represents the intensity of the blue component of the color to use when drawing this control part.
- Part identifier, and the red, green, and blue color components for the next control part. Specify color components for every part of this control whose color you want to change. If a part is not listed in the control color table, the Dialog Manager draws it in its default color.

Figure 6-51 shows the format of a text style table.

Figure 6-51 Structure of a compiled text style table



The text style table must be 20 bytes long, as shown in Figure 6-51. Multiple editable text and static text items can use the same text style record. To display text in the standard typeface, color, font size, and font style, set the item data and item offset elements for the item to 0. Allocate space for all fields in the text style table, even if they are not used.

A text style table consists of the following elements (see *Inside Macintosh: Text* for a discussion of font families, font style, and point sizes):

- Typeface. This is the name of the font family to use. If bit 15 in the item data element is set to 1, then this element contains an offset (in bytes) to a font name element at the end of the resource. If bit 0 in the item data element is set to 1, then this element contains the number of a font family. If bit 0 in the item data element is set to 0, this element is set to 0, and the system default font is used.
- Font style. This is the font style to use. If bit 1 in the item data element is set to 1, then this element uses the bits of the low-order byte to describe which styles to apply to the text. If all bits in the low-order byte are set to 0, the plain font style is used. The bit numbers and the styles they represent are

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

- Font size. This is the point size of the font. If bit 2 in the item data element is set to 1, this element contains a value representing a point size. If bit 4 in the item data element is set to 1, this element contains a value to add to the current point size of the text. If bit 0 in the item data element is set to 0, this element is set to 0, and the system font size (12) is used.
- Text red color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the red component of the color to use when drawing the text.
- Text green color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the green component of the color to use when drawing the text.
- Text blue color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the blue component of the color to use when drawing the text.
- Background red color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the red component of the color to use when drawing the background behind the text.

- Background green color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the green component of the color to use when drawing the background behind the text.
- Background blue color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the blue component of the color to use when drawing the background behind the text.
- Mode. If bit 14 in the item data element is set to 1, this element contains an integer that represents how characters are placed in the bit image. The values that the Dialog Manager interprets and the constants that represent them are listed here. See *Inside Macintosh: Imaging* for a discussion of source transfer modes.

Constant	Value
`scrOr`	1
`srcXor`	2
`srcBic`	3

Summary of the Dialog Manager

Pascal Summary

Constants

```
CONST
{checking for AppendDITL, ShortenDITL, CountDITL using Gestalt function}
   gestaltDITLExtAttr = 'ditl';  {Gestalt selector for AppendDITL, etc.}
   gestaltDITLExtPresent = 0;    {if this bit's set, then AppendDITL, }
                                 { ShortenDITL, & CountDITL are available}

{item types for GetDialogItem, SetDialogItem}
   ctrlItem       = 4;     {add this constant to the next four constants}
   btnCtrl        = 0;     {standard button control}
   chkCtrl        = 1;     {standard checkbox control}
   radCtrl        = 2;     {standard radio button}
   resCtrl        = 3;     {control defined in a control resource}
   helpItem       = 1;     {help balloons}
   statText       = 8;     {static text}
   editText       = 16;    {editable text}
   iconItem       = 32;    {icon}
   picItem        = 64;    {QuickDraw picture}
   userItem       = 0;     {application-defined item}
   itemDisable    = 128;   {add to any of the above to disable it}

{item numbers of OK and Cancel buttons in alert boxes}
   ok       = 1;  {first button is OK button}
   cancel   = 2;  {second button is Cancel button}

{resource IDs of alert box icons}
   stopIcon    = 0;
   noteIcon    = 1;
   cautionIcon = 2;

{constants used for theMethod parameter in AppendDITL}
   overlayDITL       = 0;  {overlay existing items}
   appendDITLRight   = 1;  {append at right}
   appendDITLBottom  = 2;  {append at bottom}
```

```
{constants for procID parameter of NewDialog, NewColorDialog}
   dBoxProc          = 1;  {modal dialog box}
   noGrowDocProc     = 4;  {modeless dialog box}
   movableDBoxProc   = 5;  {movable modal dialog box}
```

Data Types

```
TYPE  DialogPtr      =  WindowPtr;
ResumeProcPtr        =  ProcPtr;
SoundProcPtr         =  ProcPtr;
ModalFilterProcPtr   =  ProcPtr;
DialogPeek           =  ^DialogRecord;
DialogRecord         =
RECORD
   window:     WindowRecord;  {dialog window}
   items:      Handle;        {item list resource}
   textH:      TEHandle;      {current editable text item}
   editField:  Integer;       {editable text item number minus 1}
   editOpen:   Integer;       {used internally}
   aDefItem:   Integer;       {default button item number}
END;

DITLMethod = Integer;
```

Dialog Manager Routines

Initializing the Dialog Manager

```
PROCEDURE InitDialogs       (resumeProc: ResumeProcPtr);
PROCEDURE ErrorSound        (soundProc: SoundProcPtr);
PROCEDURE SetDialogFont     (fontNum: Integer); {also spelled SetDAFont}
```

Creating Alerts

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
FUNCTION Alert              (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION StopAlert          (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION NoteAlert          (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION CautionAlert       (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION GetAlertStage      : Integer;
PROCEDURE ResetAlertStage;
```

Creating and Disposing of Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
FUNCTION GetNewDialog       (dialogID: Integer; dStorage: Ptr;
                             behind:  WindowPtr): DialogPtr;
FUNCTION NewColorDialog     (dStorage: Ptr; boundsRect: Rect; title:
                             Str255; visible: Boolean; procID: Integer;
                             behind: WindowPtr; goAwayFlag: Boolean;
                             refCon: LongInt; items: Handle): DialogPtr;
FUNCTION NewDialog          (dStorage:  Ptr; boundsRect: Rect; title:
                             Str255; visible: Boolean; procID: Integer;
                             behind: WindowPtr; goAwayFlag: Boolean;
                             refCon: LongInt; items: Handle):  DialogPtr;
PROCEDURE CloseDialog       (theDialog: DialogPtr);
PROCEDURE DisposeDialog     (theDialog: DialogPtr);
```

Manipulating Items in Alert and Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE GetDialogItem     (theDialog: DialogPtr; itemNo: Integer;
                             VAR itemType: Integer; VAR item: Handle;
                             VAR box: Rect);
PROCEDURE SetDialogItem     (theDialog: DialogPtr; itemNo: Integer;
                             itemType: Integer; item: Handle; box: Rect);
PROCEDURE HideDialogItem    (theDialog: DialogPtr; itemNo: Integer);
PROCEDURE ShowDialogItem    (theDialog: DialogPtr; itemNo: Integer);
FUNCTION FindDialogItem     (theDialog: DialogPtr; thePt: Point): Integer;
PROCEDURE AppendDITL        (theDialog: DialogPtr; theDITL: Handle;
                             theMethod: DITLMethod);
PROCEDURE ShortenDITL       (theDialog: DialogPtr; numberItems: Integer);
FUNCTION CountDITL          (theDialog: DialogPtr): Integer;
```

Handling Text in Alert and Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE ParamText         (param0: Str255; param1: Str255;
                             param2: Str255; param3: Str255);
PROCEDURE GetDialogItemText (item: Handle; VAR text: Str255);
PROCEDURE SetDialogItemText (item: Handle; text: Str255);
PROCEDURE SelectDialogItemText
                            (theDialog: DialogPtr; itemNo: Integer;
                             strtSel: Integer; endSel: Integer);
PROCEDURE DialogCut         (theDialog: DialogPtr);
PROCEDURE DialogCopy        (theDialog: DialogPtr);
```

```
PROCEDURE DialogPaste       (theDialog: DialogPtr);
PROCEDURE DialogDelete      (theDialog: DialogPtr);
```

Handling Events in Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE ModalDialog       (filterProc: ModalFilterProcPtr; VAR itemHit:
                             Integer);
FUNCTION IsDialogEvent      (theEvent: EventRecord): Boolean;
FUNCTION DialogSelect       (theEvent: EventRecord; VAR theDialog:
                             DialogPtr; VAR itemHit: Integer): Boolean;
PROCEDURE DrawDialog        (theDialog: DialogPtr);
PROCEDURE UpdateDialog      (theDialog: DialogPtr; updateRgn: RgnHandle);
```

Application-Defined Routines

```
PROCEDURE MyItem            (theWindow: WindowPtr; itemNo: Integer);
PROCEDURE MyAlertSound      (soundNo: Integer);
FUNCTION MyEventFilter      (theDialog: DialogPtr; VAR theEvent:
                             EventRecord; VAR itemHit: Integer): Boolean;
```

C Summary

Constants

```
enum {
/*checking for AppendDITL, ShortenDITL, CountDITL using Gestalt function*/
   #define gestaltDITLExtAttr 'ditl' /*Gestalt selector*/
   gestaltDITLExtPresent = 0     /*if this bit's set, then AppendDITL, */
                                 /* ShortenDITL, & CountDITL are available*/
};

enum {
/*item types for GetDItem, SetDItem*/
   ctrlItem       = 4,  /*add this constant to the next four constants*/
   btnCtrl        = 0,  /*standard button control*/
   chkCtrl        = 1,  /*standard checkbox control*/
   radCtrl        = 2,  /*standard radio button*/
   resCtrl        = 3,  /*control defined in a control resource*/
   statText       = 8,  /*static text*/
   editText       = 16, /*editable text*/
   iconItem       = 32, /*icon*/
```

```
   picItem        = 64, /*QuickDraw picture*/
   userItem       = 0,  /*application-defined item*/
   helpItem       = 1,  /*help balloons*/
   itemDisable    = 128,/*add to any of the above to disable it*/

/*item numbers of OK and Cancel buttons in alert boxes*/
   ok       = 1,  /*first button is OK button*/
   cancel   = 2,  /*second button is Cancel button*/

/*resource IDs of alert box icons*/
   stopIcon    = 0,
   noteIcon    = 1,
   cautionIcon = 2
};

enum {
/*constants used for theMethod parameter in AppendDITL*/
   overlayDITL       = 0,  /*overlay existing items*/
   appendDITLRight   = 1,  /*append at right*/
   appendDITLBottom  = 2   /*append at bottom*/
};

enum {
/*constants for procID parameter of NewDialog, NewColorDialog*/
   dBoxProc          = 1,  /*modal dialog box*/
   noGrowDocProc     = 4,  /*modeless dialog box*/
   movableDBoxProc   = 5   /*movable modal dialog box*/
};
```

Data Types

```
typedef WindowPtr DialogPtr;

typedef struct DialogRecord DialogRecord;
typedef struct DialogRecord *DialogPeek;

struct DialogRecord{
      WindowRecord   window;     /*dialog window*/
      Handle         items;      /*item list resource*/
      TEHandle       textH;      /*current editable text item*/
      short          editField;  /*editable text item number minus 1*/
      short          editOpen;   /*used internally*/
      short          aDefItem;   /*default button item number*/
};
```

```
typedef pascal void (*ResumeProcPtr)(void);
typedef pascal void (*SoundProcPtr)(void);
typedef pascal Boolean (*ModalFilterProcPtr)(DialogPtr theDialog,
                                    EventRecord *theEvent, short *itemHit);
typedef short DITLMethod;
```

Dialog Manager Routines

Initializing the Dialog Manager

```
pascal void InitDialogs     (ResumeProcPtr resumeProc);
pascal void ErrorSound      (SoundProcPtr soundProc);
pascal void SetDialogFont   (short fontNum); /*also spelled SetDAFont*/
```

Creating Alerts

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal short Alert          (short alertID, ModalFilterProcPtr filterProc);
pascal short StopAlert      (short alertID, ModalFilterProcPtr filterProc);
pascal short NoteAlert      (short alertID, ModalFilterProcPtr filterProc);
pascal short CautionAlert   (short alertID, ModalFilterProcPtr filterProc);
#define GetAlertStage()     (* (short*) 0x0A9A);
pascal void ResetAlertStage (void);
```

Creating and Disposing of Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal DialogPtr GetNewDialog
                            (short dialogID, void *dStorage,
                             WindowPtr behind);
pascal DialogPtr NewColorDialog
                            (void *dStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short procID, WindowPtr behind,
                             Boolean goAwayFlag, long refCon, Handle items);
pascal DialogPtr NewDialog
                            (void *dStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short procID, WindowPtr behind,
                             Boolean goAwayFlag, long refCon,
                             Handle items);
pascal void CloseDialog     (DialogPtr theDialog);
pascal void DisposeDialog   (DialogPtr theDialog);
```

Manipulating Items in Alert and Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void GetDialogItem   (DialogPtr theDialog, short itemNo,
                             short *itemType, Handle *item, Rect *box);
pascal void SetDialogItem   (DialogPtr theDialog, short itemNo, short
                             itemType, Handle item, const Rect *box);
pascal void HideDialogItem  (DialogPtr theDialog, short itemNo);
pascal void ShowDialogItem  (DialogPtr theDialog, short itemNo);
pascal short FindDialogItem (DialogPtr theDialog, Point thePt);
pascal void AppendDITL      (DialogPtr theDialog, Handle theDITL,
                             DITLMethod theMethod);
pascal void ShortenDITL     (DialogPtr theDialog, short numberItems);
pascal short CountDITL      (DialogPtr theDialog);
```

Handling Text in Alert and Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void ParamText       (ConstStr255Param param0,
                             ConstStr255Param param1,
                             ConstStr255Param param2,
                             ConstStr255Param param3);
pascal void GetDialogItemText
                            (Handle item, Str255 text);
pascal void SetDialogItemText
                            (Handle item, ConstStr255Param text);
pascal void SelectDialogItemText
                            (DialogPtr theDialog, short itemNo,
                             short strtSel, short endSel);
pascal void DialogCut       (DialogPtr theDialog);
pascal void DialogCopy      (DialogPtr theDialog);
pascal void DialogPaste     (DialogPtr theDialog);
pascal void DialogDelete    (DialogPtr theDialog);
```

Handling Events in Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void ModalDialog     (ModalFilterProcPtr filterProc, short *itemHit);
pascal Boolean IsDialogEvent(const EventRecord *theEvent);
pascal Boolean DialogSelect (const EventRecord *theEvent,
                             DialogPtr *theDialog, short *itemHit);
pascal void DrawDialog      (DialogPtr theDialog);
pascal void UpdateDialog    (DialogPtr theDialog, RgnHandle updateRgn);
```

Application-Defined Routines

```
pascal void MyItem          (WindowPtr theWindow, short itemNo);
pascal void MyAlertSound    (short soundNo);
pascal Boolean MyEventFilter(DialogPtr theDialog, *EventRecord theEvent,
                             *short itemHit);
```

Assembly-Language Summary

Data Structures

DialogRecord Data Structure

0	dWindow	156 bytes	window record for the alert box or dialog box
156	items	long	handle to the item list resource for the alert box or dialog box
160	teHandle	long	handle to the current editable text item
164	editField	word	current editable text item
166	editOpen	word	used internally
168	aDefItem	word	item number of the default button

Global Variables

DAStrings	Handles to text strings specified with the ParamText procedure
DABeeper	Address of current sound procedure
DlgFont	Font number for text in dialog boxes and alert boxes
ACount	Alert stage number (0 through 3) of the last alert
ANumber	Resource ID of last alert
ResumeProc	Address of resume procedure (should not be used in System 7)