`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:

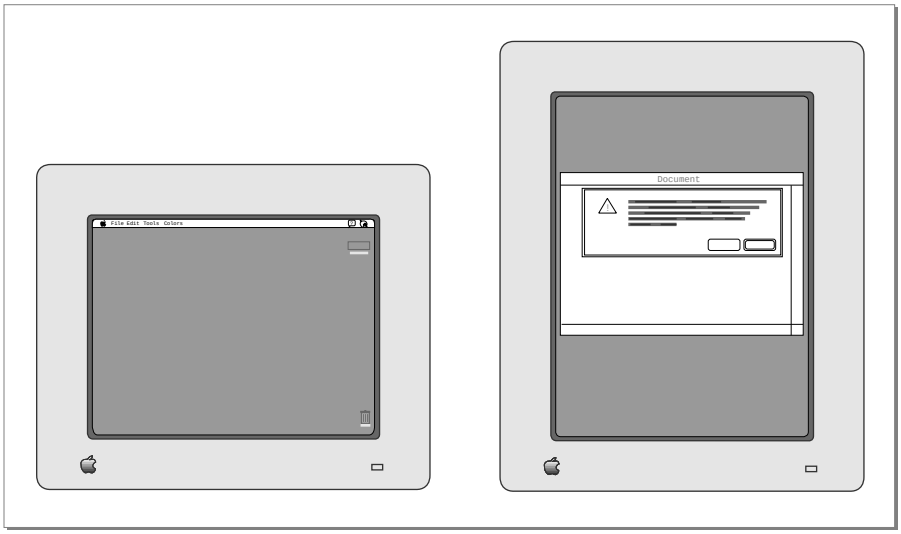
Constant	Description
`alertPositionParentWindow`	Position the alert or dialog box over the frontmost window
`alertPositionMainScreen`	Position the alert or dialog box on the main screen
`alertPositionParentWindowScreen`	Position the alert or dialog box on the screen containing the frontmost window

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.

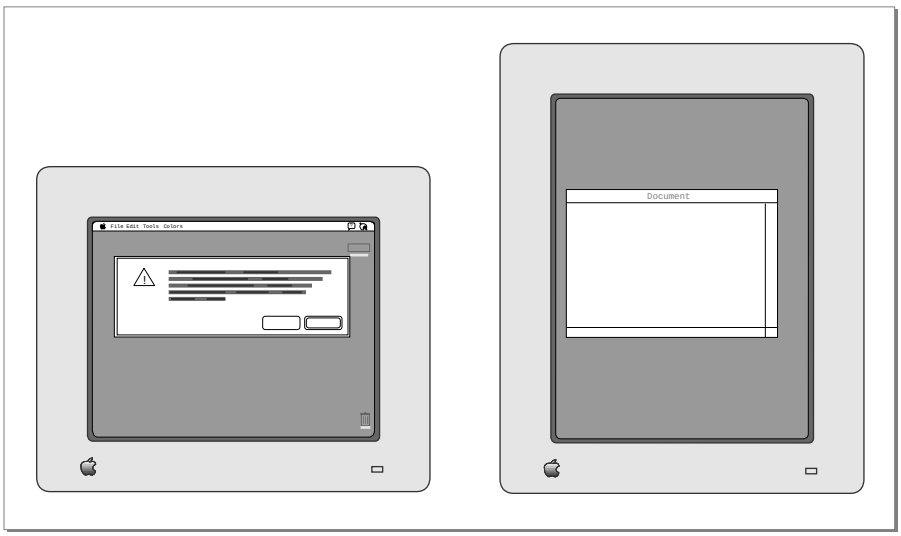
Figure 6-33 shows an alert box displayed in response to an error made by the user while working on a document; the alert resource specifies the `alertPositionParentWindow` constant, which tells the Dialog Manager to position the alert box over the frontmost window so that the window's title bar appears. 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



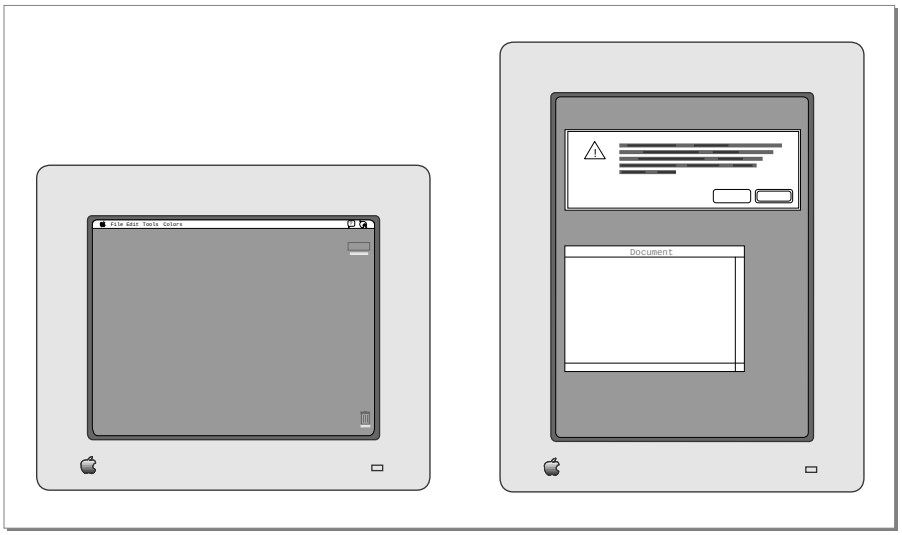
Not all alert boxes or dialog boxes relate to the frontmost window. Some may relate only to actions the user performs on the main screen. For example, Figure 6-34 illustrates an alert box displayed when the user chooses the About command from the Apple menu. For an alert box or dialog box such as this, 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.

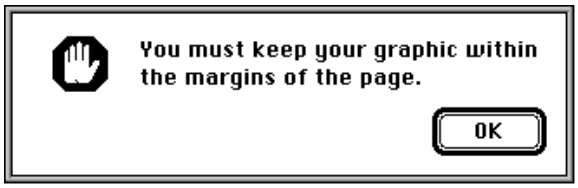
Listing 6-18 Deactivating the front window before displaying an alert box

```
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.

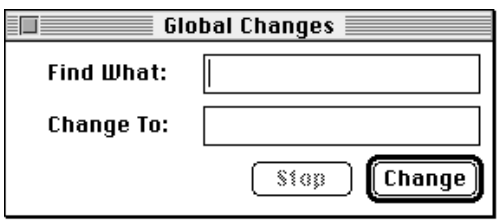
Listing 6-19 Using `GetAlertStage` to determine when to deactivate the front window

```
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

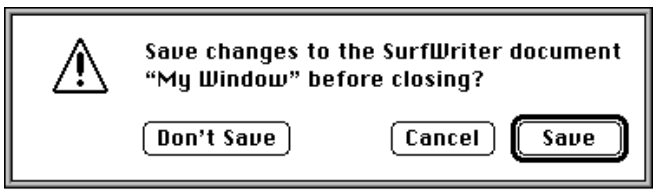
After displaying an alert box or playing an alert sound, the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions call the `ModalDialog` procedure to handle events automatically for you.

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

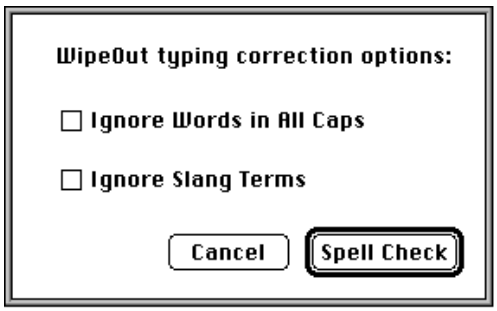
Call the `ModalDialog` procedure immediately after displaying a modal dialog box. This procedure repeatedly handles events inside the modal dialog box until an event involving an enabled item—such as a click in a radio button—occurs. If the event is a mouse-down event outside the content region of the 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.)

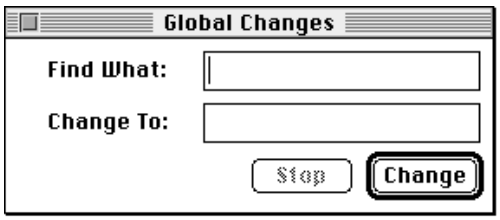
If you choose to determine whether events involve movable modal or modeless dialog boxes without the aid of the `IsDialogEvent` function, your application should be prepared to handle the following mouse events:

- 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