

## Dialog Boxes

This chapter describes how your application can use the Dialog Manager to create and manage dialog boxes. You can use dialog boxes to alert the user to unusual situations or to solicit information from the user. The Venn Diagrammer application uses one modeless dialog box and two modal dialog boxes.

This chapter shows how to

- create resources describing dialog boxes and the items in dialog boxes
- open those resources to display a dialog box
- define application-specific dialog items
- handle events associated with both modeless and modal dialog boxes

Most Macintosh applications support a number of dialog boxes and provide more complete event handling in those dialog boxes than is illustrated in this chapter. For example, the dialog boxes supported by the Venn Diagrammer application do not contain text fields. For a complete description of the capabilities of the Dialog Manager and for code samples illustrating more advanced dialog handling, see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

## About Dialog Boxes

A **dialog box** is a window that’s used for some special, limited purpose. In the simplest case, you can use a dialog box just to display information to the user. The information might be a report of some error, a greeting, or a progress bar showing what percentage of some operation has completed. Figure 7-1 shows a simple modal dialog box of this ilk; this is the box Venn Diagrammer displays when the user chooses the About Venn Diagrammer command from the Apple menu.

**Figure 7-1** An About box



This kind of dialog box is said to be **modal**: it puts the user in the state or “mode” of being able to work only inside the dialog box. To dismiss the dialog box, the user must click one or the other of the two buttons.

## Dialog Boxes

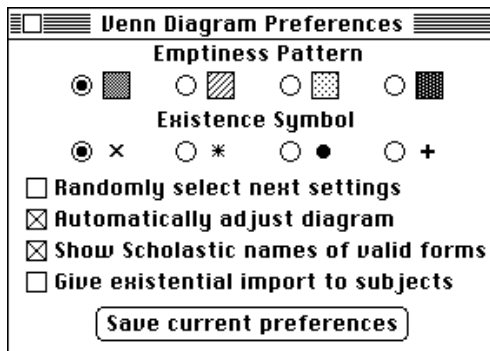
The system software distinguishes a special category of modal dialog boxes, called *alert boxes*. You'll use alert boxes to report errors or to give warnings to the user. Figure 7-2 shows an alert box. (Venn Diagrammer displays this alert box if it cannot read the resources it uses to create menus; see Listing 8-1 on page 155.)

**Figure 7-2** An alert box



Other types of dialog boxes both display information to the user and allow the user to enter or change information. You might, for instance, use a dialog box of this sort in an application that allows users to specify a word to be searched for. The Venn Diagrammer application displays the modeless dialog shown in Figure 7-3 when the user chooses the Preferences command from the Venn menu.

**Figure 7-3** A Preferences dialog box



This modeless dialog box contains a *button*, four *checkboxes*, and eight *radio buttons*. It also contains eight application-defined items—the icons used to show the available existence symbols and emptiness patterns.

In contrast to the modal dialog boxes shown in Figure 7-1 and Figure 7-2, the dialog box shown in Figure 7-3 is said to be *modeless*: the user can switch to another window or perform other actions without dismissing the dialog box. The user doesn't have to change any preferences settings or click any buttons to be able to switch to a document window or pull down a menu. Moreover, clicking a button in the modeless dialog box

## Dialog Boxes

should not dismiss it; instead, the dialog box should remain on the desktop so that the user can continue to see the information displayed in it or repeat any actions it permits.

**IMPORTANT**

To give users maximum control and minimum frustration, you should, whenever possible, implement your dialog boxes as modeless dialog boxes. ▲

The distinctive feature of dialog boxes—as opposed to windows—is that they are very easy to create and manage. The Dialog Manager looks in dialog resources to find descriptions of the dialog box and the items in it. Then the Dialog Manager draws the dialog box and handles user actions in the dialog box accordingly. This can be especially useful for managing dialog boxes that contain editable text fields. The Dialog Manager calls `TextEdit` to handle all the standard text-editing operations such as cutting, pasting, and copying.

To create a dialog box, you first need to define a *dialog resource* and a dialog *item list*. The dialog resource specifies, among other things, the rectangle on the screen in which the dialog box is drawn, a *window definition ID* indicating the type of dialog box to draw, and a resource ID of the dialog item list. A dialog resource is of type 'DLOG'. See Figure 3-2 on page 58 for the ResEdit form of a dialog resource and Listing 3-1 on page 57 for the Rez form of the same dialog resource. Both of these correspond to the dialog box in Figure 7-3.

One of the main pieces of information in a dialog resource is the resource ID of a dialog item list (a resource of type 'DITL'). The item list specifies the items—such as buttons and static text—to display in an alert box or a dialog box. (Once again, you can specify an item list graphically using a utility like ResEdit or textually in the Rez resource description language.) The Dialog Manager uses the item list both to draw the dialog box and also to handle user actions in dialog boxes. It reports user actions to your application by specifying the *item number* of the relevant item. An item's number is simply its rank in the item list. In Listing 7-1, the Venn Diagrammer application defines a number of constants to keep track of the numbers of the items in its Preferences dialog box.

**Listing 7-1** Dialog item numbers

```
iEmpty1Radio      = 1;
iEmpty2Radio      = 2;
iEmpty3Radio      = 3;
iEmpty4Radio      = 4;
iEmpty1Icon       = 5;
iEmpty2Icon       = 6;
iEmpty3Icon       = 7;
iEmpty4Icon       = 8;
iExist1Radio      = 9;
iExist2Radio      = 10;
```

## Dialog Boxes

```

iExist3Radio          = 11;
iExist4Radio          = 12;
iExist1Icon           = 13;
iExist2Icon           = 14;
iExist3Icon           = 15;
iExist4Icon           = 16;
iGetNextRandomly      = 19;
iAutoAdjust           = 20;
iShowSchoolNames      = 21;
iUseExistImport        = 22;
iSaveVennPrefs         = 23;

```

**Note**

Notice that several item numbers (namely, 17 and 18) are missing from this list. They are the item numbers of the two text labels “Emptiness Pattern” and “Existence Symbol.” Venn Diagrammer ignores those item numbers because clicking them has no effect. ♦

Dialog boxes can contain various sorts of items, such controls (buttons, checkboxes, and radio buttons) and fields for entering and editing text. The Dialog Manager recognizes these constants for dialog box items:

```

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

```

Several Dialog Manager routines return these constants to your application. For instance, you can get information about a particular dialog item by calling the `GetDialogItem` routine:

```
GetDialogItem(myDialog, itemNum, myType, myHand, myRect);
```

Suppose, for example, that `itemNum` has the value specified by the constant `iSaveVennPrefs`. Then on return from the procedure call, `myType` will contain the value `ctrlItem+btnCtrl`, indicating that the specified item is a standard button control.

## Dialog Boxes

As you can see, a dialog box can contain standard user interface elements like buttons, checkboxes, icons, and even arbitrary pictures. If you need to include other kinds of elements in a dialog box, you can create application-defined items. Because the Dialog Manager uses the constant `userItem` to designate these items, they're often called *user items*. The Venn Diagrammer application employs eight user items in the Preferences dialog box, to draw the four emptiness patterns and the four existence symbols.

When you use any application-defined user items in a dialog box, your application needs to tell the Dialog Manager how to draw the items and what to do in response to user selections of those items. See “Setting Up Application-Defined Items” beginning on page 139 for instructions on implementing user items in a dialog box.

### Note

Most dialog boxes don't need to contain user items. The Venn Diagrammer application uses them because it needs to draw bit images (not entire icons) in the dialog box. ♦

## Using Modeless Dialog Boxes

---

To display a modeless dialog box, you can create the dialog box by calling `GetNewDialog`. Then you can respond to user actions in the dialog box by intercepting dialog-related events in your main event loop and handling those events. The Dialog Manager calls the Control Manager to draw any controls you've put in the dialog box and handle user actions in them. If the dialog box contains any application-defined user items, you need to provide the Dialog Manager with a drawing procedure so that it knows how to draw the items. You also need to handle user actions for any such application-defined items yourself.

### Creating a Modeless Dialog Box

---

You can create a modeless dialog box by calling `GetNewDialog` and passing it the resource ID of an appropriate 'DLOG' resource. The Venn Diagrammer application supports only one modeless dialog box, in which the user can set various application preferences. Venn Diagrammer displays that dialog box after the user chooses the Preferences command from the Venn menu.

```
iGetVennPrefs:
    DoModelessDialog(rVennDPrefsDial, gPrefsDialog);
```

As you can see, Venn Diagrammer simply calls the application-defined procedure `DoModelessDialog`, passing it a resource ID specifying the dialog box to open and a global variable in which to return the dialog pointer created by `GetNewDialog`. Listing 7-2 defines the `DoModelessDialog` procedure.

## Dialog Boxes

**Listing 7-2** Creating a modeless dialog box

```

PROCEDURE DoModelessDialog (myKind: Integer; VAR myDialog: DialogPtr);
  VAR
    myPointer: Ptr;
BEGIN
  IF myDialog = NIL THEN                                {the dialog box doesn't exist yet}
    BEGIN
      myPointer := NewPtr(sizeof(DialogRecord));
      IF myPointer = NIL THEN
        exit(DoModelessDialog);

      myDialog := GetNewDialog(myKind, myPointer, WindowPtr(-1));
      IF myDialog <> NIL THEN
        BEGIN
          DoSetupUserItems(myKind, myDialog);           {set up user items}
          DoSetupCtrlValues(myDialog);                   {set up initial values}
        END;
      END
    END
  ELSE
    BEGIN
      ShowWindow(myDialog);
      SelectWindow(myDialog);
      SetPort(myDialog);
    END;
END;
END;

```

The `DoModelessDialog` procedure first determines whether the specified dialog box has already been created, by checking the value of the global variable passed to it. If the variable contains any value other than `NIL`, the dialog box already exists (but is perhaps hidden or obscured by other windows). If so, `DoModelessDialog` simply makes the dialog box visible (by calling `ShowWindow`), makes it the active window (by calling `SelectWindow`), and establishes it as the current graphics port (by calling `SetPort`).

If, however, the specified dialog box doesn't exist yet, then `DoModelessDialog` allocates memory for a new dialog record and (if successful) calls `GetNewDialog`, passing it the appropriate resource ID. If `GetNewDialog` returns successfully (as indicated by a returned dialog pointer whose value isn't `NIL`), `DoModelessDialog` then calls two application-defined routines, `DoSetupUserItems` and `DoSetupCtrlValues`, to tell the Dialog Manager how draw the user items in the dialog box and to set the correct initial values for the dialog box's radio buttons and checkboxes.

## Setting Up Application-Defined Items

Whenever a modeless dialog box contains application-defined user items, you need to tell the Dialog Manager how to draw them. You do this by calling the Dialog Manager procedure `SetDialogItem` for each application-defined item in the dialog box. Listing 7-3 shows the `DoSetupUserItems` procedure called by `DoModelessDialog` (defined in Listing 7-2).

**Listing 7-3**      Setting up application-defined dialog items

```
PROCEDURE DoSetupUserItems (myKind: Integer; VAR myDialog: DialogPtr);
    VAR
        myType:      Integer;
        myHand:      Handle;
        myRect:      Rect;
        count:       Integer;
        origPort:    GrafPtr;
BEGIN
    GetPort(origPort);
    SetPort(myDialog);

    CASE myKind OF
        rVennDPrefsDial:
            FOR count := 1 TO kVennPrefsItemCount DO
                IF count IN [iExist1Icon..iExist4Icon,
                           iEmpty1Icon..iEmpty4Icon] THEN
                    BEGIN
                        GetDialogItem(myDialog, count, myType, myHand, myRect);
                        SetDialogItem(myDialog, count, myType, @DoUserItem, myRect);
                    END;
                OTHERWISE
                    ;
            END;

    SetPort(origPort);
END;
```

The `DoSetupUserItems` procedure simply selects the relevant application-defined items, retrieves information about each item (by calling `GetDialogItem`), and then calls `SetDialogItem` to associate a particular application-defined drawing procedure with each item. As you can see, the drawing procedure (`DoUserItem`) is the same for each user item in the Preferences dialog box. This is possible because the Dialog

## Dialog Boxes

Manager passes the drawing procedure the dialog pointer and item number when it wants a particular item to be drawn. Listing 7-4 defines the Venn Diagrammer procedure that draws user items.

---

**Listing 7-4** Drawing application-defined dialog items

```
PROCEDURE DoUserItem (myDialog: DialogPtr; myItem: Integer);
  VAR
    myType:      Integer;
    myHand:      Handle;
    myRect:      Rect;
    origPort:    GrafPtr;
BEGIN
  GetPort(origPort);
  SetPort(myDialog);

  GetDialogItem(myDialog, myItem, myType, myHand, myRect);

  IF myDialog = gPrefsDialog THEN
    CASE myItem OF
      iExist1Icon..iExist4Icon:
        BEGIN
          DoPlotIcon(myRect, GetIcon(kExistID + myItem - iExist1Icon),
                     myDialog, srcCopy);
        END;
      iEmpty1Icon..iEmpty4Icon:
        BEGIN
          DoPlotIcon(myRect, GetIcon(kEmptyID + myItem - iEmpty1Icon),
                     myDialog, srcCopy);
          FrameRect(myRect);
        END;
    OTHERWISE
      ;
    END; {CASE}

  SetPort(origPort);           {restore original port}
END;
```

The `DoUserItem` procedure is also fairly simple. It makes sure that the dialog pointer passed to it picks out the Preferences dialog box. Then it calls the application-defined procedure `DoPlotIcon` (defined in Listing 5-8 on page 101) to draw the appropriate part of an icon in the item rectangle. If the emptiness patterns are being drawn, `DoUserItem` also draws a box around the pattern (by calling `FrameRect`).

## Handling User Actions in a Modeless Dialog Box

The Venn Diagrammer application calls its `DoHandleDialogEvent` function for each event it retrieves from the Event Manager. Its strategy is to determine if the returned event applies to a dialog box. If so, `DoHandleDialogEvent` handles the event and returns `TRUE` to indicate that it did so; otherwise, `DoHandleDialogEvent` just returns `FALSE` to indicate that it didn't handle the event. Listing 7-5 defines `DoHandleDialogEvent`. (See Listing 4-4 on page 77 to see when `DoHandleDialogEvent` is called.)

**Listing 7-5** Handling events in a modeless dialog box

```
FUNCTION DoHandleDialogEvent (myEvent: EventRecord): Boolean;
VAR
    eventHandled: Boolean;           {did we handle the event?}
    myDialog: DialogPtr;
    myItem: Integer;
BEGIN
    eventHandled := FALSE;
    IF FrontWindow <> NIL THEN
        IF IsDialogEvent(myEvent) THEN
            IF DialogSelect(myEvent, myDialog, myItem) THEN
                BEGIN
                    eventHandled := TRUE;
                    SetPort(myDialog);

                    IF myDialog = gPrefsDialog THEN
                        BEGIN
                            CASE myItem OF
                                iEmpty1Radio..iEmpty4Radio:
                                    gEmptyIndex := myItem;
                                iEmpty1Icon..iEmpty4Icon:
                                    gEmptyIndex := myItem - 4;
                                iExist1Radio..iExist4Radio:
                                    gExistIndex := myItem - iEmpty4Icon;
                                iExist1Icon..iExist4Icon:
                                    gExistIndex := myItem - (iEmpty4Icon + 4);
                                iGetNextRandomly:
                                    gStepRandom := NOT gStepRandom;
                                iAutoAdjust:
                                    gAutoAdjust := NOT gAutoAdjust;
                                iShowSchoolNames:
                                    gShowNames := NOT gShowNames;
                                iUseExistImport:
```

## Dialog Boxes

```

        gGiveImport := NOT gGiveImport;
        iSaveVennPrefs:
            DoSavePrefs;
        OTHERWISE
            ;
    END;

    DoSetupCtrlValues(myDialog);           {update values}
END;

DoHandleDialogEvent := eventHandled;
END;
```

The `DoHandleDialogEvent` function calls the Dialog Manager's `IsDialogEvent` function to determine whether at the time of the event the frontmost window is a dialog box. If not, then `DoHandleDialogEvent` just exits and returns the value `FALSE`. If, however, the event did occur while a dialog box was active, then the event might apply to that dialog box. To determine whether it does apply, `DoHandleDialogEvent` calls the Dialog Manager's `DialogSelect` function, which handles most of the events relating to a dialog box. For example, if the event is an update or activate event for the dialog box, `DialogSelect` updates or activates the dialog box and returns `FALSE` (to indicate that no further processing is required by the calling application).

If the event involves an enabled item in the dialog box, `DialogSelect` returns a function result of `TRUE`. In the `myItem` parameter, it returns the item number of the item selected by the user. In the `myDialog` parameter, it returns a pointer to the dialog record for the dialog box where the event occurred. In all other cases, the `DialogSelect` function returns `FALSE`. When `DialogSelect` returns `TRUE`, you should do whatever is appropriate as a response to the event involving that item in that particular dialog box; when it returns `FALSE`, you should do nothing.

The `DoHandleDialogEvent` function uses a very simple technique for handling user selections of items in the Preferences dialog box. As you can see, it sets the appropriate application global variables for clicks of the radio buttons, and it toggles the appropriate global variables for clicks of the checkboxes. Then `DoHandleDialogEvent` calls the application-defined procedure `DoSetupCtrlValues` to change the values of those controls, turning the radio buttons and checkboxes off or on, as appropriate. Listing 7-6 gives the definition of `DoSetupCtrlValues`.

---

**Listing 7-6**      Setting the state of radio buttons and checkboxes

```

PROCEDURE DoSetupCtrlValues (myDialog: DialogPtr);
VAR
    count:      Integer;
    myType:     Integer;
```

## Dialog Boxes

```

myHand:      Handle;
myRect:      Rect;
origPort:    GrafPtr;
BEGIN
  IF myDialog = NIL THEN
    exit(DoSetupCtrlValues);

  GetPort(origPort);           {save the current graphics port}
  SetPort(myDialog);           {always do this before drawing}
  ShowWindow(myDialog);

  IF myDialog = gPrefsDialog THEN
    BEGIN
      FOR count := 1 TO kVennPrefsItemCount DO
        BEGIN
          GetDialogItem(myDialog, count, myType, myHand, myRect);
          IF myType = ctrlItem + radCtrl THEN
            CASE count OF
              iExist1Radio..iExist4Radio:
                SetCtlValue(ControlHandle(myHand),
                  ORD(gExistIndex = count - (iExist1Radio - 1)));
              iEmpty1Radio..iEmpty4Radio:
                SetCtlValue(ControlHandle(myHand),
                  ORD(gEmptyIndex = count - (iEmpty1Radio - 1)));
            OTHERWISE
              ;
          END;
          IF myType = ctrlItem + chkCtrl THEN
            CASE count OF
              iGetNextRandomly:
                SetCtlValue(ControlHandle(myHand),
                  ORD(gStepRandom = TRUE));
              iShowSchoolNames:
                SetCtlValue(ControlHandle(myHand),
                  ORD(gShowNames = TRUE));
              iUseExistImport:
                SetCtlValue(ControlHandle(myHand),
                  ORD(gGiveImport = TRUE));
              iAutoAdjust:
                SetCtlValue(ControlHandle(myHand),
                  ORD(gAutoAdjust = TRUE));
            OTHERWISE
              ;

```

## Dialog Boxes

```

                                END ;
                                END ;
                                END ;

SetPort(origPort);              {restore the previous graphics port}
END ;

```

The `DoSetupCtrlValues` procedure simply calls the Control Manager procedure `SetCtlValue` to set the value of each control in the dialog box according to the value of some global variable. This makes it easy to toggle checkboxes and to group radio buttons in such a way that exactly one radio button in each group is on.

**IMPORTANT**

The strategy for handling dialog box events described in this section might not be the best or most efficient strategy for your application. For a more complete discussion of handling dialog box events, see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*. ▲

## Using Modal Dialog Boxes

---

Remember that a modal dialog box puts the user into the state or “mode” of being able to work only inside the dialog box. The user cannot move the dialog box and can dismiss it only by clicking its buttons (perhaps after supplying some necessary information).

**Note**

The Dialog Manager also provides *movable modal dialog boxes*; these are modal dialog boxes that contain a title bar so that the user can drag the dialog box. You should use movable modal dialog boxes whenever the user might need to move a modal dialog box to see what it obscures or whenever you want allow the user to switch to another application while the dialog box is displayed. ♦

In general, it’s easier to create and handle simple modal dialog boxes than it is to create and handle modeless dialog boxes. The reason is that the Dialog Manager provides special routines that you can call to display alerts and other simple dialog boxes. The Dialog Manager also provides the `ModalDialog` procedure, which you can call to manage all user actions in modal dialog boxes.

**IMPORTANT**

Ease of implementation is not a sufficient reason for using modal dialog boxes instead of modeless ones. You should avoid using modal dialog boxes except when absolutely necessary. ▲

## Displaying a Modal Dialog Box

Listing 7-7 shows a standard way to display a modal dialog box. It defines the procedure `DoAboutBox`, which is called after the user chooses the About Venn Diagrammer command from the Apple menu.

**Listing 7-7**      Displaying a modal dialog box

```
PROCEDURE DoAboutBox (myWindow: WindowPtr);
    VAR
        myWindow:    WindowPtr;
        myDialog:    DialogPtr;
        myItem:      Integer;
    BEGIN
        myWindow := FrontWindow;
        IF myWindow <> NIL THEN
            DoActivate(myWindow, 1 - activeFlag);

        myDialog := GetNewDialog(rAboutDial, NIL, WindowPtr(-1));
        IF myDialog <> NIL THEN
            BEGIN
                SetPort(myDialog);
                DoDefaultButton(myDialog);

                REPEAT
                    ModalDialog(@MyModalFilter, myItem);
                UNTIL myItem = iOK;

                DisposeDialog(myDialog);
                SetPort(myWindow);
            END;
        END;
    END;
```

When you display a modal dialog box, you should first deactivate any existing front window. The `DoAboutBox` procedure retrieves a window pointer to the front window and passes that pointer to the application-defined activate routine `DoActivate`. Then `DoAboutBox` calls `GetNewDialog` to open the dialog box specified by the resource ID `rAboutDial`:

```
CONST
    rAboutDial = 7000;           {resource ID of About dialog}
```

If `GetNewDialog` returns a dialog pointer whose value is not `NIL`, then `DoAboutBox` calls `SetPort` to establish the new dialog box as the current drawing port. Then it calls the application-defined procedure `DoDefaultButton` (defined in Listing 7-8) to draw a

## Dialog Boxes

thick border around the default button. This indicates that the user can dismiss the dialog box by pressing the Return key or the Enter key.

---

**Listing 7-8** Outlining the default button of a modal dialog box

```
PROCEDURE DoDefaultButton (myDialog: DialogPtr);
    VAR
        myType:      Integer;
        myHand:      Handle;
        myRect:      Rect;
    BEGIN
        GetDialogItem(myDialog, iOK, myType, myHand, myRect);
        DoOutlineControl(myHand);
    END;
```

The `DoDefaultButton` procedure simply calls the application-defined procedure `DoOutlineControl` to outline the dialog item whose item number is 1 (identified by the constant `iOK`). See page 200 for a definition of `DoOutlineControl`.

At this point, the modal dialog box is displayed on the screen. The `DoAboutBox` procedure loops indefinitely, repeatedly calling `ModalDialog` until the user clicks the OK button. The `ModalDialog` procedure handles all mouse, keystroke, and update events that occur inside the dialog box until an event involving an enabled dialog item occurs. When that happens, `ModalDialog` exits and returns the dialog item number in the second parameter. Your application can then do whatever is appropriate in response to an event in that item. In `DoAboutBox`, `ModalDialog` is called repeatedly until a click in the OK button occurs. At that time, the modal dialog is removed from the screen, and `DoAboutBox` calls `SetPort` to reinstate the original drawing port.

## Defining a Modal Dialog Filter Function

---

The actions of `ModalDialog` are guided by the *modal dialog filter function* whose address is passed in its first parameter. If you pass `NIL` as the first parameter to the `ModalDialog` procedure, you'll get the standard event filtering provided by the Dialog Manager. The standard event filter function returns `TRUE` and causes `ModalDialog` to return item number 1 (the number of the default button) when the user presses the Return or the Enter key.

For most modal dialog boxes, the standard modal dialog filter function is too simple. Your application should define a modal dialog filter function that performs the following tasks:

- return `TRUE` and the item number for the default button if the user presses the Return key or the Enter key
- return `TRUE` and the item number for the Cancel button if the user presses the Escape key or the Command-period combination

## Dialog Boxes

- allow background applications to receive update events and return FALSE when they do
- return FALSE for all other events that your event filter doesn't handle

Listing 7-9 defines a modal dialog filter function that accomplishes these tasks. In addition, the filter function `MyModalFilter` handles any disk-inserted events that occur while the modal dialog box is displayed.

**Listing 7-9** A modal dialog filter function

```
FUNCTION MyModalFilter (myDialog: DialogPtr; VAR myEvent: EventRecord;
                        VAR myItem: Integer): Boolean;

VAR
    myType:      Integer;
    myHand:      Handle;
    myRect:      Rect;
    myKey:        Char;
    myIgnore:    LongInt;
BEGIN
    MyModalFilter := FALSE;           {assume we don't handle the event}

    CASE myEvent.what OF
        updateEvt:
            BEGIN
                IF WindowPtr(myEvent.message) <> myDialog THEN
                    DoUpdate(WindowPtr(myEvent.message));
                                {update the window behind}
            END;
        keyDown, autoKey:
            BEGIN
                myKey := char(BAnd(myEvent.message, charCodeMask));

                {if Return or Enter pressed, do default button}
                IF (myKey = kReturn) OR (myKey = kEnter) THEN
                    BEGIN
                        GetDialogItem(myDialog, iOK, myType, myHand, myRect);
                        HiliteControl(ControlHandle(myHand), 1);
                                {make button appear to have been pressed}
                        Delay(kVisualDelay, myIgnore);
                        HiliteControl(ControlHandle(myHand), 0);
                        MyModalFilter := TRUE;
                        myItem := iOK;
                    END;
            END;
    END;
```

## Dialog Boxes

```

    {if Escape or Cmd-. pressed, do Cancel button}
    IF (myKey = kEscape)
      OR ((myKey = kPeriod)
        AND (BAnd(myEvent.modifiers, CmdKey) <> 0)) THEN
      BEGIN
        GetDialogItem(myDialog, iCancel, myType, myHand, myRect);
        HiliteControl(ControlHandle(myHand), 1);
        {make button appear to have been pressed}
        Delay(kVisualDelay, myIgnore);
        HiliteControl(ControlHandle(myHand), 0);
        MyModalFilter := TRUE;
        myItem := iCancel;
      END;
    END;
diskEvt:
  BEGIN
    DoDiskEvent(myEvent);
    MyModalFilter := TRUE;           {show we've handled the event}
  END;
OTHERWISE
  ;
END; {CASE}
END;

```

An interesting part of `MyModalFilter` is the way it intercepts key-down events and translates them into button clicks. When, for instance, it detects that the Return key was pressed, it calls `GetDialogItem` to retrieve a handle to the first item in the item list (by convention, the OK button). Then `MyModalFilter` calls `HiliteControl` to invert the state of the button, waits for a specified number of ticks, and then calls `HiliteControl` once again to restore the button to its original state. Finally, it sets the function result and the variable parameter `myItem`, thus informing the calling routine that the event was handled.