6

Version 1.2 (Frozen)

DIALOGS AND ALERTS

Includes Demonstration Program DialogsAndAlertsPascal

Introduction

Alerts and Alert Boxes

Alerts, which may be an alert sound or an alert box or both, warn the user whenever an unusual or potentially undesirable situation occurs within your application. An alert box, unlike a dialog box, typically requires only the user's acknowledgment in order for your application to proceed.

Dialog Boxes

Dialog boxes allow the user to provide additional information or to modify settings before your application carries out a command.

Because it greatly simplifies the task, the Dialog Manager should be used to implement alerts and simple dialog boxes. However, it is sometimes desirable to bypass the Dialog Manager and use Window Manager, Control Manager, QuickDraw, and Event Manager routines to create and manage complex dialog boxes. Some situations which tend to diminish the advantages of using the Dialog Manager are:

- The dialog box contains more than 20 items.
- You need a multi-part control, such as a scroll bar.
- You need to display a moving indicator, such as a progress indicator.
- You need to display a list in the dialog box. (See Chapter 18 — Lists and Custom List Definition Functions .)
- You need to display text in a font other than the system font.
- Your application must respond to events other than mouse-down events, key-down events inside editable text items, and a limited number of keyboard equivalent key-down events.

The two issues to consider in relation to the creation and management of dialog boxes are therefore:

- Whether to use the Window Manager and Control Manager, instead of the Dialog Manager, to create the dialog box.
- Whether to use the Event Manager, Window Manager, Control Manager, and TextEdit, instead of the Dialog Manager, to handle events.

In addressing these issues, you should also bear in mind that a hybrid approach, in which the Dialog Manager is used to create, but not manage, a dialog box, is also possible.

Types of Alerts, Alert Boxes, and Dialog Boxes

Types of Alerts

When an alert condition occurs, and depending on the nature of that condition, your application can simply play an alert sound or it can display an alert box. Your application can also base its response on the number of consecutive times the condition occurs, possibly playing an alert sound at first and subsequently displaying an alert box.

Alert Sound

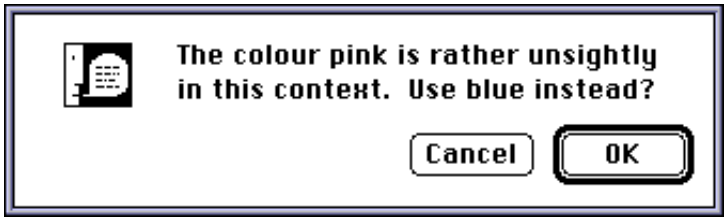
The **system alert sound** is a sound resource stored in the System file. It is played whenever the system software or your application uses the Sound Manager routine `SysBeep`. The alert sound should be used for errors which are minor and immediately obvious, such as attempting to backspace past the left boundary of a text field.

Alert Boxes

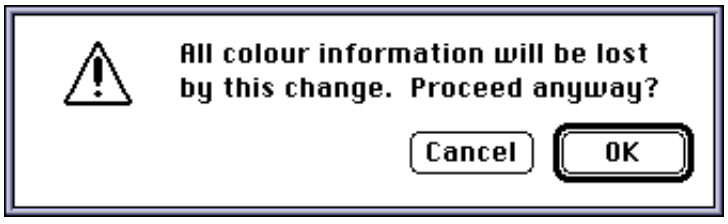
There are three standard types of alert boxes, all of which are illustrated at Fig 1:

- **Note Alert.** The note alert is used to inform users of an occurrence which will not have disastrous consequences. Usually, a note alert simply offers information. Sometimes, as shown at Fig 1, a note alert may ask a simple question and provide a choice of responses.

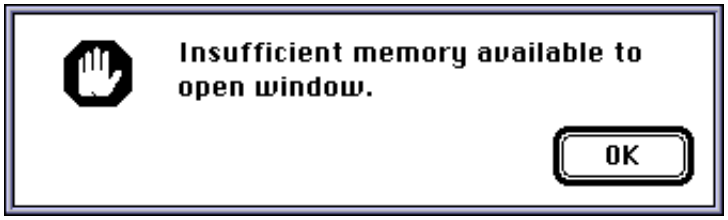
- **Caution Alert.** The caution alert is used to alert the user to an operation which may have undesirable results if it is allowed to continue. As shown at Fig 1, you should provide the user, via the buttons, with a choice of whether to continue or stop the action.



NOTE ALERT



CAUTION ALERT



STOP ALERT

FIG 1 - TYPES OF ALERTS

- **Stop Alert.** The stop alert is used to inform the user that a problem or situation is so serious that the action cannot be completed. As shown at Fig 1, stop alerts typically have only an OK button.

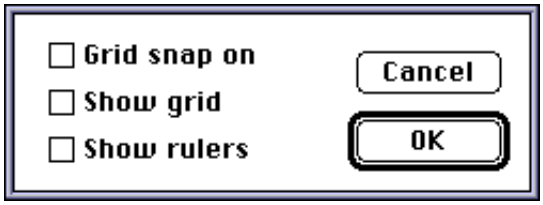
The icons in the examples at Fig 1 are supplied automatically by the system.

Custom Alert Boxes

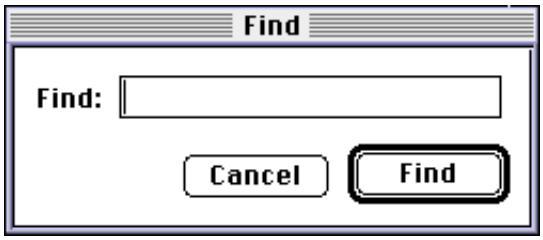
You can also create **custom alert boxes**, which might contain your own icons (or, possibly, no icons). Custom alert boxes are typically used for **About...** boxes.

Types Of Dialogs Boxes

There are three types of dialog boxes, all of which are illustrated in the examples at Fig 2:



MODAL DIALOG BOX



MOVABLE MODAL DIALOG BOX

MODELESS DIALOG BOX

FIG 2 - TYPES OF DIALOG BOXES

Modal Dialog Boxes

Fixed-position modal dialog boxes place the user in the state, or mode, of being able to work only inside the dialog box. The only response the user receives when clicking outside the dialog box is the alert sound. This type of dialog box looks like an alert box except that it may contain other types of controls in addition to buttons.

Movable Modal Dialog Boxes

Movable modal dialog boxes retain the modal characteristic of their fixed-position counterparts, the main difference being the addition of a title bar which enables the user to drag the dialog box so as to uncover obscured areas of an underlying window. The other difference is that this type of dialog allows the user to bring another application to the front by clicking in one of the application's windows or by choosing the application's name from the Application menu.

The absence of close boxes and zoom boxes in the title bar visually suggests to the user that the dialog box is modal.

Modeless Dialog Boxes

Modeless dialog boxes look like document windows and do not require the user to respond before doing anything else. The user should be able to move the dialog box, activate and deactivate it, and close it like any document window; however, unlike document windows, the box should contain no scroll bars and no size box.

When you display a modeless dialog box, your application must allow the user to perform other operations without first dismissing the dialog. When the user clicks a button in the dialog box, the application should not remove the dialog; it should only be removed by a click in the close box or when the user selects Quit from the File menu.

Because of the difficulty of revoking the last action invoked from a modeless dialog box, it typically does not have a Cancel button, although it may have a Stop button to halt long operations such as searching and printing.

Items in Alert and Dialog Boxes

You use resources called **item lists** to specify the **items** to appear in alert boxes and dialog boxes. Alert boxes should usually contain only informative text, button controls and perhaps a graphic (that is, an icon or QuickDraw picture). Dialog boxes may contain the following items:

- Informative or instructional text.
- Rectangles in which text may be entered (that is, **editable text items**).
- Controls.
- Graphics (that is, icons or QuickDraw pictures).
- Other items as defined by your application (for example, status bars).

Enabled and Disabled Items

Items may be enabled or disabled. An enabled item is one for which the Dialog Manager reports user-initiated events. A disabled item is one for which the Dialog Manager does not report events. Your application can enable and disable any item.

Note that a *disabled item* is not the same as an *inactive control.* The distinction is as follows:

- **Disabled Item.** When you do not want the Dialog Manager to report clicks in a control, you disable the item. Note that the Dialog Manager makes no visual distinction between a disabled item and an enabled item.
- **Inactive Control.** When you do not want the Control Manager to respond to clicks in a control, you make it inactive with the Control Manager routine `HiliteControl`. The Control Manager displays an inactive control in a way which indicates that it is not active (that is, by dimming it).

Default Buttons in Alert Boxes

To assist the user who is not sure how to respond when an alert appears, your application specifies a **default button** for every alert box. In alert boxes, the Dialog Manager draws a bold outline around this button. If the user presses the Return key or the Enter key, the Dialog Manager acts as if the user had clicked the default button.

Default Buttons in Dialog Boxes

Dialog boxes typically contain an OK button and a Cancel button, although the OK button may sometimes contain a title reflecting the action to be performed. The default button requirement (that is, the response to the Return and Enter key) also applies to dialog boxes.

Unless you provide your own event filter function (see below), the Dialog Manager treats the first button item in the dialog as the default button. Note, however, that the Dialog Manager does not draw a bold outline around the default button in dialog boxes.

Removal of Alert and Dialog Boxes

The Dialog Manager automatically removes an alert box when the user clicks any enabled item.

Your application should remove a modal or movable modal dialog box only after the user clicks one of its enabled buttons.

Your application should not remove a modeless dialog box unless the user clicks its close box or chooses **Close** from the **File** menu when the modeless dialog box is the active window.

Creating Alerts

`Alert`, `NoteAlert`, `CautionAlert` and `StopAlert` are used to create alerts. Icons associated with the latter three automatically appear in the upper-left corner of the alert boxes. The `Alert` function allows you to display no icon or your own icon. When the user clicks a button in the alert box, the functions return the button's item number and close the alert box.

`Alert`, `NoteAlert`, `CautionAlert` and `StopAlert` take descriptive information about the alert from an `'ALRT'` resource. The ID of this resource is passed in the function's first parameter.

The `'ALRT'` Resource

The `'ALRT'` resource ID is the first parameter in the `Alert`, `NoteAlert`, `CautionAlert` and `StopAlert` call. A typical `'ALRT'` resource, in Rez input format, is as follows:

```
resource 'ALRT' (kSaveAlertID, purgeable)
{
  {94,80,183,438},/* Rectangle for alert box. */
  kAlertItemList, /* Resource ID for item list ('DITL') resource. */
  {   /* ALERT STAGES:
    OK, visible, sound1,  /* 4th alert stage. */
    OK, visible, sound1,  /* 3rd alert stage. */
    OK, visible, sound1,  /* 2nd alert stage. */
    OK, visible, sound1,  /* 1st alert stage. */
  },
  alertPositionParentWindow /* Positioning constant. */
};
```

Alert Stages

When an alert condition occurs, your application can base its response on the number of times that condition has occurred. In the example `'ALRT'` resource definition above, the listing specifies that each consecutive time the user repeats the action which invokes the alert, the Dialog Manager should outline the OK button and treat it as the default button, display the alert box and play a single system alert sound.

You can, however, define different responses for each of the four alert stages. This is most appropriate for stop alerts — that is, those which signify that an action cannot be completed, especially when that action has a high probability of being accidental. In such circumstances, your application might simply

play the alert sound the first two times the user makes the mistake and, subsequently, display the alert box as well. Note that every occurrence of the mistake after the fourth is treated as a fourth stage alert.

Specifying `invisible` in the alert stages section of the resource definition causes the alert box not to be displayed. Specifying `sound2` or `sound3` causes the system alert sound to be played twice and three times respectively.[1] [2]

Positioning Constant

If a positioning constant is not provided, the Dialog Manager places the alert box at the global coordinates you specify for the alert's rectangle.

Event Filter Function

The second parameter in `Alert`, `NoteAlert`, `CautionAlert` and `StopAlert` calls is a pointer to an **event filter function**. Specifying `nil` for the event filter function parameter causes the functions to use the **standard event filter function**, which provides for users pressing the Return or Enter keys in lieu of clicking on the default button.

The standard event filter function, however, has some basic limitations. The main limitation is that it does not permit background applications to receive or respond to update events. For that reason, your application should provide a replacement event filter function (see below) which, in addition to allowing users to press the Return or Enter keys in lieu of clicking on the default button, and as a minimum, allows background applications to receive null events.

Window Definition ID

When you create an alert box, the Dialog Manager always passes to the Window Manager the window definition ID represented by the constant `dBoxProc`.[3]

Creating Dialog Boxes

`GetNewDialog` or `NewDialog` are used to create dialog boxes. `GetNewDialog` is usually used, since it takes information about the dialog box from a `'DLOG'` resource. `GetNewDialog` creates a **dialog record** from the information in the `'DLOG'` resource and returns a pointer to that record.

If `NULL` is specified as the second parameter in the `GetNewDialog` call, `GetNewDialog` itself creates a non-relocatable block for the dialog record. Passing `NULL` is appropriate for modal and movable modal dialog boxes; however, in order to avoid heap fragmentation effects, you should ordinarily allocate your own memory for modeless dialog box dialog records (just as you would for a window record) and specify the pointer to that memory block in the second parameter to the `GetNewDialog` call.[4]

The Dialog Record

The dialog record created by the `GetNewDialog` call is defined by the data type `DialogRecord`:

```
type
  DialogRecord = record
                 window:          WindowRecord;
                 items:           Handle;
                 textH:           TEHandle;
                 editField:       integer;
                 editOpen:        integer;
                 aDefItem:        integer;
```

[1] If the user has set the speaker volume to 0, the menu bar blinks once in place of each sound.

[2] If you want the Dialog Manager to play sounds other than the system sound, you must write your own sound procedure and then call `ErrorSound`, passing it a pointer to your sound procedure. This makes your sound procedure the current sound procedure.

[3] The Window Manager always displays an alert box in front of all other windows.

[4] However, see Footnote 8 at Chapter 4 — Widows.

```
  end;

  DialogPeek = ^DialogRecord;
```

Note that the dialog record includes a window record field. The Dialog Manager sets the `windowKind` field of this window record to `dialogKind`.

The `'DLOG'` Resource

An example of a `'DLOG'` resource, in Rez input format, is as follows:

```
resource 'DLOG' (kSpellCheckID, purgeable)
{
  {62,184,216,448},         /* Rectangle for dialog box. */
  dBoxProc,                 /* Window definition ID for modal dialog box. */
  visible,                  /* Display this dialog box immediately. */
  noGoAway,                 /* No go away box. (Use goAway for modeless dialog box.) */
  0x0,                      /* Initial reference constant is 0. */
  kSpellCheckDITL,          /* Item list ('DITL') resource ID */
  "SpellCheck Options",     /* Title, (Use empty string for modal dialogs.) */
  alertPositionParentWindow /* Positioning constant. */
};
```

Window Definition ID. In this example, the window definition ID represented by the constant `dBoxProc` is specified, meaning that the resource is for a modal dialog box. The window definition IDs you use for dialog boxes are as follows:

Type of Dialog Box	Window definition ID
Modal dialog box	`dBoxProc`
Movable modal dialog box	`movableDBoxProc`
Modeless dialog box	`noGrowDocProc`

Visible/Invisible. The `visible` constant specifies that the dialog box will be displayed immediately. If `invisible` is specified, a call to `ShowWindow` is required to display the dialog box when required.

Reference Constant. The `0x0` specified as the reference constant is simply a filler. You may wish to store a number which represents the dialog box type, or perhaps a handle to a record which maintains state information about the dialog box.

Positioning Constant. Other options for the positioning constant are `alertPositionParentScreen` and `alertPositionParentWindowScreen`.

Items for Alerts and Dialog Boxes

The `'DITL'` Resource

You use an **item list** (`'DITL'`) **resource** to store information about all the items in an alert or dialog box. The `'DITL'` resource ID is specified in the associated `'ALRT'` or `'DLOG'` resource.

Within a `'DITL'` resource for an alert box you can specify static text, buttons, icons and QuickDraw pictures. In dialog boxes, checkboxes, buttons, editable text and controls may be added.

An example of a `'DITL'` resource, in Rez input format, is as follows:

```
resource 'DITL' (kAboutBoxDITL, purgeable) /* Items for About… alert box */
{
                        /* ITEM NO 1 */
  { {86,201,106,259},   /* Display rectangle for item (Local to the dialog box.) */
    Button     {        /* Item is a button. */
      enabled,          /* Enable item. (Return clicks.) */
      "OK"              /* Title for button */
    },
```

```
                        /* ITEM NO 2 */
      {10,20,42,52},    /* Display rectangle for item. */
      Icon       {      /* Item is an icon. */
        disabled,       /* Disable item.  (Do not return clicks.) */
        kAboutIconID    /* 'ICON' or 'cicn' resource ID. */
      },
                        /* ITEM NO 3 */
      {10,78,74,259},   /* Display rectangle for item. */
      StaticText {      /* Item is static text */
        disabled,       /* Disable item.  (Do not return clicks.) */
        "My Application\n" /* Text string to display */
        "Version 1.0"
      },
                        /* ITEM NO 4 */
      {0,0,0,0},        /* (Help items get an empty rectangle.) */
      HelpItem   {      /* Invisible item for reading in help balloons. */
        disabled,       /* Disable item.  (Do not return clicks.) */
        HMScanhdlg      /* Scan resource type 'hdlg' for help balloons. */
        {kAboutBoxHelp} /* Get 'hdlg' resource with this resource ID. */
      }
    }
};
```

Note that, as in this example, `'DITL'` resources should invariably be marked as purgeable.

Items are usually referred to by their position in the item list, that is, by their **item number**. In the example, the Dialog Manager would return 1 when the user clicked in the OK button.

As previously stated, `GetNewDialog` creates a dialog record. It then reads in the `'DITL'` resource and stores a handle to it in the dialog record. Because the Dialog Manager always makes a copy of the `'DITL'` resource and uses that copy, several independent dialog boxes may use the same `'DITL'` resource. `AppendDITL` and `ShortenDITL` may be used to modify or customise copies of a shared item list resource for use in individual dialog boxes.

Display Rectangles

The **display rectangle** determines the location of an item within an alert box or dialog box.

Controls. For controls, the display rectangle becomes the control's **enclosing rectangle**. To match a control's enclosing rectangle to its display rectangle, specify an enclosing rectangle in the `'CNTL'` resource which is identical to the display rectangle specified in the `'DITL'` resource.[5]

Editable Text Items. For an editable text item, the display rectangle becomes the TextEdit **destination rectangle** and **view rectangle** (see Chapter 17 — Text and TextEdit). Word wrapping occurs within display rectangles that are large enough to contain multiple lines of text, and the text is clipped if there is more text than will fit in the rectangle. The Dialog Manager draws a rectangle three pixels outside the display rectangle.

Static Text Items. For a static text item, the Dialog manager draws the text within the display rectangle just as it draws editable text items, except that the framed rectangle is not drawn.

Icons and QuickDraw Pictures. For an icon or QuickDraw picture larger than the display rectangle, the Dialog Manager scales the icon or picture to fit the display rectangle.

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

Conventions for Positioning Button and Text Display Rectangles

Recommended locations for buttons and text in an alert box are illustrated at Fig 3.

[5] When an item is a control defined in a control resource, the rectangle added to the update region is the rectangle defined in the `'CNTL'` resource, not the display rectangle specified in the `'DITL'` resource.

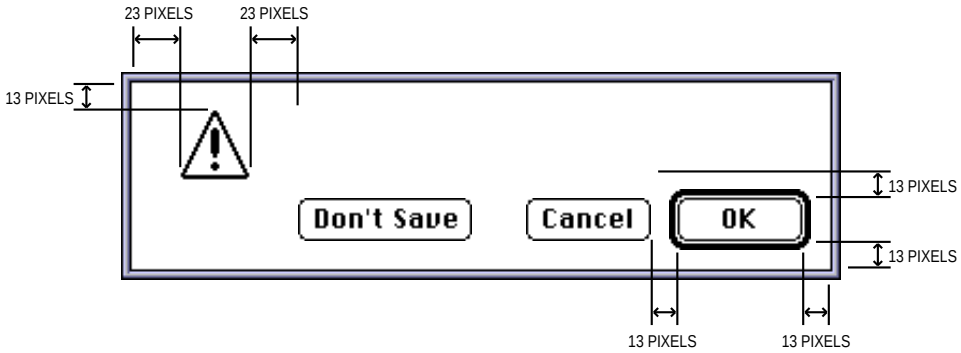


FIG 3 - CONSISTENT SPACING OF BUTTONS AND TEXT IN AN ALERT BOX

Be aware that the Window Manager adds three white pixels inside the window frame when it draws alert boxes and modal dialog boxes. Therefore, specify display rectangle locations as follows when you use tools like Rez and ResEdit:

- Place the lower-right button 10 pixels from the right edge and 10 pixels from the bottom edge of the alert or modal dialog box. Align the display rectangles for other bottom-most and right-most items with this button.

- Place the upper-left icon 10 pixels from the top edge and 20 pixels from the left of the alert or modal dialog box. Align the display rectangles for the other top-most and left-most items with this item. (The Dialog Manager automatically places the note, caution and stop icons in this position.)

- Place other elements 13 or 23 pixels apart, as shown at Fig 3.

Item Types

The example `'DITL'` resource contains four item types. The following shows the full range of item types you can include in alert and dialog boxes :

Constant	Description	
`StaticText`	Static text, that is, text that cannot be edited.	
`Button`	Button control.	
`Icon`	Icon whose black and white resource is stored in an `'ICON'` resource and whose colour version is stored in a `'cicn'` resource with the same ID as the `'ICON'` resource.	
`Picture`	QuickDraw picture stored in a `'PICT'` resource.	
`HelpItem`	Invisible item which makes the Help Manager associate help balloons with the other items defined in the item list resource.	
`RadioButton`	Radio button control.	(Use in dialog boxes only.)
`CheckBox`	Check box control.	(Use in dialog boxes only.)
`Control`	Control defined in a `'CNTL'` resource.	(Use in dialog boxes only.)
`EditText`	Editable text item.	(Use in dialog boxes only.)
`UserItem`	Application-defined item.	(Use in dialog boxes only.)

Note that static and editable text is drawn, by default, using the system font; however, the font used to draw this text can be changed using `SetDialogFont`.

Default Buttons

Default Button in Alert Boxes

The first item in an alert box's item list should always be the OK button. If a Cancel button is necessary, it should be the second item.

Default Button in Dialog Boxes

As previously stated, the Dialog Manager does not automatically draw a bold outline around the default button for dialog boxes. You should normally give every dialog a default button.[6] If you do not provide your own event filter function, the Dialog Manager treats the first item in the item list resource as the default button.

Enabling and Disabling Items

Generally, you should enable controls only. You typically disable icons, pictures and static text items because there is no requirement to receive reports of mouse-down events in these items.

Editable text items are normally disabled because an editable text item is *not* a control and your application does not need to respond to clicks in the item.

Editable Text Items

Editable text items accept input from the keyboard. The Dialog Manager automatically displays the insertion point caret in an editable text item to indicate that it is accepting keyboard input. If you do not want to display default text in an editable text item, specify an empty string as the item's final element in the `'DITL'` resource. Specify a string if you want to display default text.[7]

The Dialog Manager handles mouse-down and Tab key-down events. If an alert or dialog box contains more than one editable text item, this enables the user to select any item by either clicking the desired item or pressing the Tab key to cycle through the available items in the sequence determined by their position in the item list. You should therefore ensure that the item numbers of editable text items in your `'DITL'` resource reflect the sequence in which you require them to be selected by successive Tab key presses.

Manipulating Items

Routines for Manipulating Items

Dialog Manager routines[8] for manipulating items are as follows:

Routine	Description
`AppendDITL`	Adds items to a dialog box.
`ShortenDITL`	Removes items from a dialog box.
`GetDialogItem`	Returns the item type, the display rectangle, and the control handle or application-defined function of a given item in a dialog box.
`SetDialogItem`	Sets the item type and the display rectangle of an item or, for application-defined items, the draw function of an item.
`GetAlertStage`	Returns the stage of the last occurrence of an alert.
`ResetAlertStage`	Resets the stage of the last occurrence of an alert.
`HideDialogItem`	Hides the given item.
`ShowDialogItem`	Re-displays a hidden item.
`GetDialogItemText`	Returns the text of an editable or static text item.
`SelectDialogItemText`	Selects the text of an editable text item.
`FindDialogItem`	Finds an item that contains a specified point within a dialog box.
`CountDITL`	Counts items in a dialog box.
`ParamText`	Substitutes up to four different text strings in static text items.

[6] However, do not display a bold outline around any button if you use the Return key in editable text items.

[7] You can use `SelIText` to indicate a selected text range within an editable text item.

[8] Note that there are alternative (older) spellings for some of these routines.

Adding Items to an Existing Dialog Box

You can dynamically add items to, and remove items from, a dialog box by using `AppendDITL` and `ShortenDITL`. These routines are especially useful where several dialog boxes share the same `'DITL'` resource and you want to add or remove items as appropriate for individual dialog boxes.

When you call `AppendDITL` , you specify a new `'DITL'` resource to append to the dialog box's existing `'DITL'` resource. You also specify where the Dialog Manager should display the new items by using one of these constants in the `AppendDITL` call:

Constant	Value	Description
`overlay`	`0`	Overlay existing items. Coordinates of the display rectangle are interpreted as local coordinates within the dialog box.
`AppendDITLRight`	`1`	Append at right. Display rectangles are interpreted as relative to the upper-right coordinate of the dialog box.
`appendDITLBottom`	`2`	Append at bottom. Display rectangles are interpreted as relative to the lower-left coordinate of the dialog box.

As an alternative to passing these constants, you can pass a negative number to `AppendDITL`, which appends the items relative to an existing item in the dialog box. The absolute value of this number is interpreted as the item in the dialog box relative to which the new items are to be positioned. For example, `-2` would cause the display rectangles of the appended items to be offset from the upper-left corner of item number 2 in the dialog box.

`AppendDITL` modifies the contents of the dialog box (for instance, by enlarging it). To use the unmodified version of the dialog box at a later time, you should call `ReleaseResource` to release the memory occupied by the appended item list.

Getting Text From Editable Text Items

Getting text from an editable text item involves a call to `GetDialogItem`, which returns a handle to the item, and passing this handle to `GetDialogItemText`.

Changing Static Text

`ParamText` may be used to change static text in an alert box or dialog box. A common example is the inclusion of the window title in static text such as "**Save changes to the document ... before closing?**". In this case, the window's title could be retrieved using `GetWTitle` and inserted by `ParamText` at the appropriate text replacement variable (`^0`, `^1`, `^2` or `^3`) specified in the text string field of the static text item in the `'DITL'` resource. (Since there are four text replacement variables, `ParamText` can supply up to four text strings for a single alert or dialog box.)

Using an Application-Defined Item to Draw a Default Button's Bold Outline

You can include your own type of **application-defined item** in a dialog box (for example, a clock). One use of an application-defined item is to draw a bold outline around the default button in a dialog box. To define this item, include an item of type `userItem` in your `'DITL'` resource. It should have a display rectangle but no text and no resource ID associated with it. The following example shows part of a `'DITL'` resource containing the item:

```
resource 'DITL' (kSpellCheckDITL, purgeable)
{
  {
    /* ITEM NO 1 - OK button (default). */
    {123,170,144,254}, Button {enabled, "OK"},
    ...
    /* ITEM NO 6 - Application-defined item */
    {115,164,152,260}, UserItem {disabled,}
  }
}
```

Note that the application-defined item is disabled because the OK button, which should lay within the application-defined item, is itself enabled.

You must then provide a routine which draws your application-defined item. Your draw routine must have two parameters: a dialog pointer and an item number from the dialog box's 'DITL' resource. The routine is installed using GetDialogItem and SetDialogItem . GetDialogItem is used to get the handle to the application-defined item specified in the 'DITL' resource. SetDialogItem is then used to replace this handle with a pointer to your draw routine.

When calling your draw routine, the Dialog Manager sets the current port to the dialog box's graphics port. The Dialog Manager then calls your routine to draw the application-defined whenever the Dialog Manager receives an update event for the dialog box.

It is best if the associated 'DLOG' resource specifies the invisible constant, making the dialog box invisible while you install the draw routine for the specified item. ShowWindow may then be called to display the dialog box.

Displaying Alert and Dialog Boxes

As previously stated, Alert, NoteAlert, CautionAlert and StopAlert are used to display alert boxes, GetNewDialog displays those dialog boxes that you specify as visible in their 'DLOG' resources, and you must use ShowWindow following the GetNewDialog call to display dialog boxes that you specify as invisible in their 'DLOG' resources. You should invariably specify (WindowPtr) -1 as a parameter to GetNewDialog so as to display a dialog box as the active (frontmost) window.

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

- Deactivate the frontmost window (if one exists).
- If you are displaying a modeless dialog box, determine whether you have previously invoked it. If so, use ShowWindow to make it visible and SelectWindow to make it active.
- Adjust your menus appropriately for a modal dialog box with editable text items and for any movable modal and modeless dialog you wish to display.

Deactivating Windows Behind Alert and Dialog Boxes

Movable Modal and Modeless Dialog Boxes

You do not have to deactivate the front window *explicitly* when displaying movable modal and modeless dialog boxes. The Event Manager continues sending your application activate events for your windows as needed, which you typically handle in your main event loop.

Alert and Modal Dialog Boxes

On the other hand, ModalDialog, which initiates the session of user interaction with alert and modal dialog boxes, traps all events before they are passed to your event loop (which, of course, ordinarily handles activate events for your windows). Thus, if a window is active, you must *explicitly* deactivate it before displaying an alert or modal dialog box.

If your application does not display an alert box during certain alert stages, use the GetAlrtStage function to test for those stages before deactivating the active window.

Adjusting Menus for Alert and Modal Dialog Boxes

The Dialog Manager and Menu Manager interact to provide varying degrees of access to the menus in your menu bar. When your application displays an alert box or modal dialog box (that is, a window of type dBoxProc), system software disables all items in the **Application** and **Help** menus except the **Show Balloons**/**Hide Balloons** command in the Help menu.

When your application displays an alert box or calls ModalDialog to display a modal dialog box, the Dialog Manager determines whether any of the following cases is true:

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

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

- The Menu Manager disables all your application's menus.
- 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 for **Cut**, **Copy** and **Paste**, the Menu Manager enables those three commands and the menu which contains them.

Alert Boxes and Modal Dialog Boxes Without Editable Text 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.

Modal Dialog Boxes with Editable Text Items

However, because system software cannot handle the **Undo** and **Clear** commands (or any other context-dependent command), 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 its first item (typically, the **About**... command) in order to take control of menu bar access away from the Dialog Manager.
- Disable all of the application's menus except the Edit menu, as well as any inappropriate commands in the Edit menu.
- Use `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the **Cut**, **Copy**, **Paste**, and **Clear** commands in editable text items.[9]
- Provide your own code for supporting the **Undo** command.
- Enable your application's items in the Help menu as appropriate.

Restoring Menus

When the user dismisses the alert box or modal dialog box, the Menu Manager restores all menus to their previous state unless your application handles its own menu bar access, in which case your application must restore the menu bar to its previous state.

Adjusting Menus for Movable Modal and Modeless Dialog Boxes

Although it always leaves the Help and Application menus and their items enabled, system software does nothing else to help 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.

Movable Modal Dialog Box

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

- Leave the Apple menu open so that the user can open other applications with it.

[9]Your application can test whether a dialog box is the front window when handling mouse down events and call these routines as appropriate.

- If your movable modal dialog box contains editable text items, use the Dialog Manager routines `DialogCut`, `DialogCopy`, `DialogPaste` and `DialogDelete` to support the **Cut**, **Copy**, **Paste**, and **Clear** commands in editable text items.

- Disable all of your other menus.

Modeless Dialog Boxes

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 routines `DialogCut`, `DialogCopy`, `DialogPaste` and `DialogDelete` to support the **Cut**, **Copy**, **Paste**, and **Clear** commands in editable text items.

Displaying Multiple Alert and Dialog Boxes

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

The Window Manager automatically dims the 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 `HiliteControl` to make the controls of the dialog inactive. You should also draw the outline of the default button in grey instead of black.

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. 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 yourself. (See Chapter 22 — Miscellany for a description of the Notification Manager).

Including Colour

On colour monitors, the Dialog Manager automatically adds the system default colours to the frame and title bar of your alert and dialogs boxes so that they match the colours of the windows, alert boxes and dialog boxes used by the system software. Colour in the content region is, however, another matter.

Alert and dialog boxes are created with a black-and-white graphics port. However, you can force the Dialog Manager to create alert and dialog boxes with a colour graphics port by providing a **dialog color table resource** (`'dctb'`) with the same resource ID as the alert or dialog resource.

There are two specific circumstances where you will want to ensure that the dialog box is created with a colour graphics port:

- When you want to produce a blended grey colour for outlining the default button when it is inactive (that is, dimmed). Unless a blended grey *colour* is used to draw the dimmed default button outline, the only alternative is to set the drawing pen pattern to the QuickDraw variable `gray`. `gray` represents a black-and-white *pattern*, not a colour. For aesthetic reasons, this is not appropriate on a colour or grey scale display.

- When you want to display a colour icon or picture in the dialog box (or alert box), and have the icon or picture appear in colour, rather than black-and-white, in a system software environment earlier than version 7.1 as updated by System Update 3.0.

When you create a 'dctb' resource, you should not change the system's default colours. The following is an example of a dialog colour table resource which leaves the default colours intact but forces the Dialog Manager to supply a colour graphics port:

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

Handling Events in Alert and Dialog Boxes

Overview

Alert and Modal Dialog Boxes

When Alert, NoteAlert, CautionAlert, and StopAlert are used to display alerts, the Dialog Manager handles all of the events generated by the user until the user clicks a button. When the user clicks a button, the functions invert the button, close the alert box and report the user's selection to the application.

The Dialog Manager routine ModalDialog initiates a session of user interaction with a modal dialog and handles most of that interaction until the user selects an item. ModalDialog then reports that the user selected an enabled item, and your application is then responsible for performing the action associated with that item. Your application typically calls ModalDialog repeatedly until the user clicks on the OK or Cancel button.

Event Filter Function. As previously stated, you should supply an event filter function for Alert boxes so as to avoid the basic limitations of the standard event filter function. This requirement also applies to modal dialog boxes. You can supply an event filter function as one of the parameters to Alert, NoteAlert, CautionAlert, StopAlert, and ModalDialog. If you supply an event filter function, these routines will pass events to your event filter function *before h*andling each event. In this way, your event filter function can handle any event not handled by the Dialog Manager.

Movable Modal and Modeless Dialog Boxes

For movable modal and modeless dialog boxes, two alternatives are available to handle events:

- Determine whether an event occurred while the dialog box was the frontmost window, perhaps using IsDialogEvent for that purpose.[10] If the dialog box was the frontmost window, use DialogSelect to:
  - Handle key-down events in editable text items automatically.
  - Handle update and activate events automatically.
  - Report the enabled items that the user clicks.[11]

  Then respond appropriately to clicks in your active items.
- Handle events in modeless and movable modal dialog boxes much as you handle events in other windows.

Responding to Events in Controls

For clicks in checkboxes, pop-up menus and radio buttons, your application should use the Control Manager routines GetControlValue and SetControlValue to get and set the item's value. When the user

[10] For every type of event which occurs while the dialog box is active, IsDialogEvent returns TRUE.

[11] DialogSelect differs from ModalDialog in that it returns control after every event, not just events related to enabled items.

clicks on the OK button, your application should perform whatever action is necessary according to the values returned by each of the checkboxes and radio buttons.

Events and Editable Text Items

Editable text items are typically disabled because you generally do not need to be informed every time the user clicks on one of them or types a character. Instead, you simply need to retrieve the text when the user clicks the OK button.

When you use `ModalDialog` or `DialogSelect`, the Dialog Manager calls TextEdit to automatically handle keystrokes and mouse actions within editable text items so that:

- When the user clicks the item, a blinking vertical bar, called the **caret**, appears.
- When the user drags over text or double-clicks a word, that text is highlighted and replaced by whatever the user types.
- When the user holds down the Shift key while clicking and dragging, the highlighted section 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 (if any), wrapping around to the first one if there are no more items.

Caret Blinking

If your movable modal or modeless dialog box contains any editable text items, you should call `DialogSelect` in your main event loop's idle processing function. This is necessary because `DialogSelect` calls `TEIdle` to make the caret blink within your editable text items when null events are received.[12]

Edit Menu

The **Edit** menu should be left enabled and you should use `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` 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.

Return Key, Enter Key, and the Default Button Outline

If you do not supply an event filter function, and the user presses the Return or Enter key while the modal dialog is on-screen, the Dialog Manager treats the event as a click on the default button regardless of whether the dialog box contains an editable text item. If you do supply an event filter function and it responds to the user pressing Return or Enter by moving the cursor in editable text items, do not display a bold outline around any buttons.

Responding to Events in Alert Boxes

After displaying an alert box or playing an alert sound, `Alert`, `NoteAlert`, `CautionAlert`, and `StopAlert` call `ModalDialog` to handle events automatically. `ModalDialog`, in turn, gets each event by calling `GetNextEvent`.

If the event is a mouse-down outside the alert box's content region, `ModalDialog` emits the system alert sound and gets the next event.

[12] You should also ensure that, when caret blinking is required, the `sleep` parameter in the `WaitNextEvent` call is set to a value no greater that that returned by `GetCaretTime`.

`ModalDialog` is continually called until the user selects an enabled control, at which time `Alert`, `NoteAlert`, `CautionAlert` and `StopAlert` remove the alert box from the screen and return the item number of the selected control. Your application then should then respond appropriately.

The standard event filter function allows users to press the Return or Enter key in lieu of clicking the default button. When you write your own event filter function (see below), it should emulate the standard filter function by responding to the keyboard in the same way. For events inside the alert box, `ModalDialog` passes the event to your event filter function *before* handling the event. Your event filter function thus provides a means to:

- Handle events which `ModalDialog` does not handle.
- Override events `ModalDialog` would otherwise handle.

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, `TrackControl` is called to track the mouse. If the user releases the mouse button while the cursor is still in the control, the alert box is removed and the control's item number is returned.
- If the user presses the mouse button while the cursor is in any enabled item other than a control, the alert box is removed and the item number is returned.
- 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, nothing happens.

Responding To Events in Modal Dialog Boxes

Your application should call `ModalDialog` immediately after displaying a modal dialog box. `ModalDialog` repeatedly handles events inside the dialog box until an event involving an enabled item occurs, at which time `ModalDialog` returns the item number. Your application should then respond appropriately to that item number. Your application should continually call `ModalDialog` until the user clicks on the OK or Cancel button, at which time your application should close the dialog box.

If the event is a mouse-down outside the content region, `ModalDialog` emits the alert sound and gets the next event.

Unless your event filter function (see below) 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, by either displaying an insertion point caret or by selecting text. If a key-down event occurs and there is an editable text item, text editing and entry are handled as previously described. If the editable text item is enabled, `ModalDialog` returns its item number after it receives either the mouse-down or key-down event.
- If the user presses the mouse button while the cursor is in a control, `TrackControl` is called. If the user releases the mouse button while the cursor is within an enabled control, `ModalDialog` returns the control's item number.
- If the user presses the mouse button while the cursor is in any other enabled item, `ModalDialog` returns the item number. (Generally, only controls should be enabled.)

- If the user presses the mouse button while the cursor is in a disabled item or no item, nothing happens.

Event Filter Functions for Alert and Modal Dialog Boxes

In early versions of the system software, when a single application controlled the computer, the standard event filter for alert and modal dialog boxes was usually sufficient. However, because the standard filter does not permit background applications to receive or respond to update events, it is no longer adequate. Your application should therefore provide a simple event filter function which performs these functions and also allows inactive windows to receive update events. In most cases, you can use the same filter function for all of your alert boxes and modal dialog boxes.

You can also use your event filter to handle events that `ModalDialog` does not handle, such as a Command-period key-down event, disk-inserted events, keyboard equivalents, and mouse-down events for application-defined items.

At a minimum, your event filter 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 combination.
- Update your windows in response to update events and return `false`. [13]
- Return `false` for all events that your event filter does not handle.

Your event filter function should have three parameters and return a Boolean value:

```
eventFilter(theDialog : DialogPtr; var theEvent : EventRecord;
                                   var itemHit : integer) : boolean;
```

When your function returns `false`, `ModalDialog` handles the event. If your function does handle the event, it should return `true` and, in the `itemHit` parameter, the number of the item that it handled. `ModalDialog` and, in turn, `Alert`, `NoteAlert`, `CautionAlert`, and `StopAlert`, then return this item number in their own `itemHit` parameter.

Because `ModalDialog` calls `GetNextEvent` with a mask which excludes disk-inserted events, your event filter function can call `SetSystemEventMask` to reset the mask to accept disk-inserted events if you wish the filter function to handle disk-inserted events.

To give visual feedback indicating which item has been selected, your filter function should invert buttons activated by keyboard equivalents. A good rule of thumb is to invert a button for eight ticks.

As previously stated, if your modal dialog box contains editable text items, your application should support the use of **Edit** menu items, in which case your filter function should test for, and handle, mouse-down events in the menu bar and key-down events for keyboard equivalents.

Mouse Events in Movable Modal and Modeless Dialog Boxes

When your application detects that an event occurred while a movable modal or modeless dialog box was the frontmost window, you should use `DialogSelect` to:

- Handle key-down events in editable text items automatically.
- Handle update and activate events automatically.
- Report the enabled items that the user clicks.

[13] This action also allows background applications to receive update events.

You must then use other ToolBox routines to handle other types of events in the dialog box. 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. (For **Edit** menu selections, you can use `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the **Cut**, **Copy**, **Paste**, and **Clear** commands in editable text items.)

- Clicks in the content region of an active movable modal or modeless dialog box. You can use `DialogSelect` 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 active by making it the front 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. For modeless dialog boxes, your application should bring the inactive window to the front.

- Mouse-down events in the title bar of an active movable modal or modeless dialog box. Your application should use `DragWindow` to move the dialog box in response to the user's actions.

- Mouse-down events in the title bar 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 dialog box, whichever action is most appropriate.

Keyboard Events in Movable Modal and Modeless Dialog Boxes

When your application detects that a keyboard event occurred while a movable modal or modeless dialog box was the frontmost window, your application should be prepared to handle the following keyboard events:

- Keyboard equivalents applicable to the dialog box, such as Command-X to perform a cut in an editable text item.

- 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 and Command-period keystrokes, to which your application should respond as if the user clicked the Cancel button.

- Key-down and auto-key events in editable text items, in response to which your application should call `DialogSelect` (which will call TextEdit to automatically handle the keystrokes).

Activate and Update Events in Movable Modal and Modeless Dialog Boxes

Your application should be prepared to handle activate and update events for both modeless and movable modal dialog boxes.

You can use `DialogSelect` to assist you in handling update and activate events. For faster performance, you may want to use the `UpdateDialog` function when handling update events. Both `DialogSelect` and `UpdateDialog` use `SetPort` to make the dialog box the current graphics port before redrawing or updating it.

You should use `HiliteControl` to make the buttons and other controls inactive in a movable modal or modeless dialog box when you deactivate it. When you activate a movable modal or modeless dialog box again, you should use `HiliteControl` to make the controls active.

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

In response to an update event, `DialogSelect` calls `BeginUpdate`, `DrawDialog` (to redraw the entire dialog box), and then `EndUpdate`. The faster alternative (`UpdateDialog`) redraws only the update region. It must be preceded by a `BeginUpdate` call and followed by an `EndUpdate` call.

Closing Dialog Boxes

Use `CloseDialog` to dispose of a dialog box if you allocated the memory for the dialog record yourself, otherwise use `DisposeDialog`.

`CloseDialog` removes a dialog from the screen and deletes it from the window list. It also releases memory occupied by the data structures associated with the dialog box, and 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. `CloseDialog` does not dispose of the dialog record or the `'DITL'` resource.

`DisposeDialog`, on the other hand, calls `CloseDialog` and, in addition, releases the memory occupied by the dialog record and item list resource.

For modeless and movable modal dialog boxes, you might find it more efficient to hide the dialog box with `HideWindow` rather than remove its structures. In that way, the dialog will remain available, and in the same location and with the same settings as when it was last used.

If you adjust the menus when you display a dialog box, be sure to return them to an appropriate state when you close the dialog box.

Main Dialog Manager Constants, Data Types and Routines

Constants

Item Types for `GetDialogItem` and `SetDialogItem`

```
ctrlItem                = 4    Add this constant to the next four constants.
btnCtrl                 = 0
chkCtrl                 = 1
radCtrl                 = 2
resCtrl                 = 3
statText                = 8
editText                = 16
iconItem                = 32
picItem                 = 64
userItem                = 0
helpItem                = 1
itemDisable             = 28   Add to any of the above to disable it.
```

Item Numbers for OK and Cancel Buttons in Alert Boxes

```
ok                      = 1
cancel                  = 2
```

New, More Standard Names For Dialog Item Constants

```
kControlDialogItem         = ctrlItem
kButtonDialogItem          = ctrlItem + btnCtrl
kCheckBoxDialogItem        = ctrlItem + chkCtrl
kRadioButtonDialogItem     = ctrlItem + radCtrl
kResourceControlDialogItem = ctrlItem + resCtrl
kStaticTextDialogItem      = statText
kEditTextDialogItem        = editText
kIconDialogItem            = iconItem
kPictureDialogItem         = picItem
```

```
kUserDialogItem           = userItem
kItemDisableBit           = itemDisable
kStdOkItemIndex           = ok
kStdCancelItemIndex       = cancel
```

Resource IDs of Alert Box Icons

```
stopIcon                  = 0
noteIcon                  = 1
cautionIcon               = 2
```

Constants Use for theMethod Parameter in AppendDITL

```
overlayDITL               = 0
appendDITLRight           = 1
appendDITLBottom          = 2
```

Constants Use for procID Parameter in NewDialog and NewColorDialog

```
dBoxProc                  = 1    Modal dialog box.
noGrowDocProc             = 4    Modeless dialog box.
movableDBoxProc           = 5    Movable modal dialog box.
```

Data Types

```
DialogPtr = WindowPtr;
DialogRef = DialogPtr;
```

Dialog Record

```
type
  DialogRecord = record
    window:     WindowRecord;
    items:      Handle;
    textH:      TEHandle;
    editField:  integer;
    editOpen:   integer;
    aDefItem:   integer;
  end;

DialogPeek = ^DialogRecord;
```

Routines

Note: Some Dialog Manager routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. The following reflects the newest spellings, as specified in version 2.1 of the Universal Interfaces.

Initialising the Dialog Manager

```
procedure  InitDialogs(ignored: UNIV Ptr);
procedure  ErrorSound(soundProc: SoundUPP);
procedure  SetDialogFont( value: integer );
```

Creating Alerts

```
function   Alert(alertID: integer; modalFilter: ModalFilterUPP): integer;
function   StopAlert(alertID: integer; modalFilter: ModalFilterUPP): integer;
function   NoteAlert(alertID: integer; modalFilter: ModalFilterUPP): integer;
function   CautionAlert(alertID: integer; modalFilter: ModalFilterUPP): integer;
function   GetAlertStage : integer;
procedure  ResetAlertStage;
```

Creating and Disposing of Dialog Boxes

```
function   GetNewDialog(dialogID: integer; dStorage: UNIV Ptr; behind: WindowRef): DialogRef;
function   NewDialog(wStorage: UNIV Ptr; var boundsRect: Rect; title: ConstStr255Param;
           visible: boolean; procID: integer; behind: WindowRef; goAwayFlag: boolean;
           refCon: longint; itmLstHndl: Handle): DialogRef;
function   NewColorDialog(dStorage: UNIV Ptr; var boundsRect: Rect; title: ConstStr255Param;
           visible: boolean; procID: integer; behind: WindowRef; goAwayFlag: boolean;
           refCon: longint; items: Handle): DialogRef;
```

```
procedure  CloseDialog(theDialog: DialogRef);
procedure  DisposeDialog(theDialog: DialogRef);
```

Manipulating Items in Alert and Dialog Boxes

```
procedure  GetDialogItem(theDialog: DialogRef; itemNo: integer; var itemType: integer;
           var item: Handle; var box: Rect);
procedure  SetDialogItem(theDialog: DialogRef; itemNo: integer; itemType: integer;
           item: Handle; var box: Rect);
procedure  HideDialogItem(theDialog: DialogRef; itemNo: integer);
procedure  ShowDialogItem(theDialog: DialogRef; itemNo: integer);
function   FindDialogItem(theDialog: DialogRef; thePt: Point): integer;
procedure  AppendDITL(theDialog: DialogRef; theHandle: Handle; method: DITLMethod);
procedure  ShortenDITL(theDialog: DialogRef; numberItems: integer);
function   CountDITL(theDialog: DialogRef): integer;
```

Handling Text in Alert and Dialog Boxes

```
procedure  ParamText(param0: ConstStr255Param; param1: ConstStr255Param;
           param2: ConstStr255Param; param3: ConstStr255Param);
procedure  GetDialogItemText(item: Handle; var text: Str255);
procedure  SetDialogItemText(item: Handle; text: ConstStr255Param);
procedure  SelectDialogItemText(theDialog: DialogRef; itemNo: integer; strtSel: integer;
           endSel: integer);
procedure  DialogCut(theDialog: DialogRef);
procedure  DialogPaste(theDialog: DialogRef);
procedure  DialogCopy(theDialog: DialogRef);
procedure  DialogDelete(theDialog: DialogRef);
```

Handling Events in Dialog Boxes

```
procedure  ModalDialog(modalFilter: ModalFilterUPP; var itemHit: integer);
function   IsDialogEvent(var theEvent: EventRecord): boolean;
function   DialogSelect(var theEvent: EventRecord; var theDialog: DialogRef;
           var itemHit: integer): boolean;
procedure  DrawDialog(theDialog: DialogRef);
procedure  UpdateDialog(theDialog: DialogRef; updateRgn: RgnHandle);
```

Demonstration Program

```
{ ############################################################################################
// DialogsAndAlertsPascal.p
// ############################################################################################
//
// This program:
//
// •  Opens a window for the purposes of displaying text and for proving correct window
//    updating and activation/deactivation in the presence of alert and dialog boxes.
//
// •  Allows the user to invoke a demonstration alert, a modal dialog box, a movable
//    modal dialog box and a modeless dialog box via the Demonstration menu.
//
// The modal dialog box contains three checkboxes.
//
// The movable modal dialog box contains three radio buttons.
//
// The modeless dialog box contains an icon and an editable text item.  The editable text
// item is supported by the Edit menu Cut, Copy, Paste and Clear commands.
//
// An application-defined event filter function is used for the alert box and modal
// dialog box.
//
// An application-defined function is used to draw the bold outline around the default
// button in the modal, movable modal and modeless dialog boxes.
//
// The program utilises the following resources:
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, Demonstration and Help
//    menus (preload, non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially visible).
//
// •  An 'ALRT' resource (purgeable).
```

```
34  //
35  // •  'DLOG' resources (purgeable) (initially not visible) and associated 'DITL'
36  //    resources (purgeable).
37  //
38  // •  'dctb' resources (purgeable) to force the Dialog Manager to create colour graphics
39  //    ports for the movable modal and modeless dialog boxes.
40  //
41  // •  A 'cicn' resource (purgeable).
42  //
43  // •  A 'SIZE' resource with the acceptSuspendResumeEvents and doesActivateOnFGSwitch,
44  //    and is32BitCompatible flags set.
45  //
46  // ############################################################################### }
47
48  program DialogsAndAlertsPascal(input, output);
49
50  { ……… include the following Universal Interfaces }
51
52  uses
53
54    Controls, Windows, Menus, Quickdraw, Fonts, Events, OSUtils, Processes, TextUtils, Dialogs,
55    TextEdit, QuickdrawText, Types, Memory, Palettes, ToolUtils, Devices, SegLoad,
56    Sound, OSUtils;
57
58  { ……… define the following constants }
59
60  const
61
62  mApple = 128;
63    iAbout = 1;
64  mFile = 129;
65    iClose = 4;
66    iQuit = 11;
67  mEdit = 130;
68    iCut = 3;
69    iCopy = 4;
70    iPaste = 5;
71    iClear = 6;
72  mDemonstration = 131;
73    iAlert = 1;
74    iModal = 2;
75    iMovable = 3;
76    iModeless = 4;
77  rMenubar = 128;
78  rNewWindow = 128;
79  rAlert = 128;
80    iOK = 1;
81    iCancel = 2;
82    iUserItem = 3;
83  rModal = 129;
84    iGridSnap = 4;
85    iShowGrid = 5;
86    iShowRulers = 6;
87  rMovable = 130;
88    iCharcoal = 4;
89    iOilPaint = 5;
90    iWaterColour = 6;
91  rModeless = 131;
92    iSearch = 1;
93    iEditText = 4;
94
95  kMovableModal = 1;
96  kModeless = 2;
97
98  kReturn = $0D;
99  kEnter = $03;
100 kEscape  = $1B;
101 kPeriod = $2E;
102
103 kMaxLong = $7FFFFFFF;
104
105 { ……… user-defined types }
106
107 type
108
109 DocRec = record
```

```
110     vScrollbarHdl, hScrollbarHdl : ControlHandle;
111     end;
112 
113   DocRecPointer = ^DocRec;
114   DocRecHandle = ^DocRecPointer;
115 
116   { ……… global variables }
117 
118   var
119 
120   gWindowPtr : WindowPtr;
121   gSleepTime : longint;
122   gDone : boolean;
123   gInBackground : boolean;
124   gGridSnap : integer;
125   gShowGrid : integer;
126   gShowRule : integer;
127   gBrushType : integer;
128   gOldBrushType : integer;
129   gModelessDlgPtr : DialogRef;
130 
131   menubarHdl : Handle;
132   menuHdl : MenuHandle;
133   docRecHdl : DocRecHandle;
134 
135   { ########################################################################## DoInitManagers }
136 
137   procedure DoInitManagers;
138 
139     begin
140     MaxApplZone;
141     MoreMasters;
142 
143     InitGraf(@qd.thePort);
144     InitFonts;
145     InitWindows;
146     InitMenus;
147     TEInit;
148     InitDialogs(nil);
149 
150     InitCursor;
151     FlushEvents(everyEvent, 0);
152     end;
153       {of procedure DoInitManagers}
154 
155   { ############################################################################### DoIdle }
156 
157   procedure DoIdle(var eventRec : EventRecord);
158 
159     var
160     myWindowPtr : WindowPtr;
161     dialogType : integer;
162     itemHit : integer;
163     ignored : boolean;
164 
165     begin
166     myWindowPtr := FrontWindow;
167     if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
168       begin
169       dialogType := WindowPeek(myWindowPtr)^.refCon;
170 
171       if (dialogType = kModeless) then
172         ignored := DialogSelect(eventRec, DialogPtr(myWindowPtr), itemHit);
173       end;
174     end;
175       {of procedure DoIdle}
176 
177   { ########################################################################## DoAdjustMenus }
178 
179   procedure DoAdjustMenus;
180 
181     var
182     myWindowPtr : WindowPtr;
183     dialogType : integer;
184     menuHdl : MenuHandle;
185 
```

```
  begin
  myWindowPtr := FrontWindow;

  if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
    begin
    dialogType := WindowPeek(myWindowPtr)^.refCon;

    case (dialogType) of

      kMovableModal:
        begin
        menuHdl := GetMenuHandle(mFile);
        DisableItem(menuHdl, 0);
        menuHdl := GetMenuHandle(mEdit);
        DisableItem(menuHdl, 0);
        menuHdl := GetMenuHandle(mDemonstration);
        DisableItem(menuHdl, 0);
        EnableItem(menuHdl, 4);
        end;

      kModeless:
        begin
        menuHdl := GetMenuHandle(mFile);
        EnableItem(menuHdl, 0);
        EnableItem(menuHdl, 4);
        menuHdl := GetMenuHandle(mEdit);
        EnableItem(menuHdl, 0);
        menuHdl := GetMenuHandle(mDemonstration);
        EnableItem(menuHdl, 0);
        DisableItem(menuHdl, 4);
        end;
      end;
        {of case statement}
  end

  else if (WindowPeek(myWindowPtr)^.windowKind = userKind) then
    begin
    menuHdl := GetMenuHandle(mFile);
    EnableItem(menuHdl, 0);
    DisableItem(menuHdl, 4);
    menuHdl := GetMenuHandle(mEdit);
    DisableItem(menuHdl, 0);
    menuHdl := GetMenuHandle(mDemonstration);
    EnableItem(menuHdl, 0);
    EnableItem(menuHdl, 4);
    end;

  DrawMenuBar;
  end;
    {of procedure DoAdjustMenus}

{ ################################################################## DoKeyDownMovableModal }

procedure DoKeyDownMovableModal(var eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  charCode : char;
  itemType : integer;
  itemHandle : Handle;
  itemRect : Rect;
  finalTicks : UInt32;

  begin
  myWindowPtr := FrontWindow;
  charCode := chr(BAnd(eventRec.message, charCodeMask));

  if ((charCode = char(kReturn)) or (charCode = char(kEnter))) then
    begin
    GetDialogItem(DialogRef(myWindowPtr), iOK, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), kControlButtonPart);
    Delay(8, finalTicks);
    HiliteControl(ControlHandle(itemHandle), 0);
    DisposeDialog(DialogRef(myWindowPtr));
    end
```

```
  else if ((charCode = char(kEscape)) or ((BAnd(eventRec.modifiers, cmdKey) <> 0)
         and (charCode = char(kPeriod)))) then
    begin
    GetDialogItem(DialogRef(myWindowPtr), iCancel, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), kControlButtonPart);
    Delay(8, finalTicks);
    HiliteControl(ControlHandle(itemHandle), 0);
    gBrushType := gOldBrushType;
    DisposeDialog(DialogRef(myWindowPtr));
    end;

  end;
    {of procedure DoKeyDownDocument}

{ ############################################################################ DoItemHitModeless }

procedure DoItemHitModeless(myDialogRef : DialogRef);

  var
  oldPort : WindowPtr;
  printRect, itemRect : Rect;
  itemType : integer;
  itemHdl : Handle;
  itemString : string;

  begin
  GetPort(oldPort);
  SetPort(gWindowPtr);

  SetRect(printRect, 15, 13, 369, 36);

  PenMode(patBic);
  PaintRect(printRect);

  GetDialogItem(myDialogRef, iEditText, itemType, itemHdl, itemRect);
  GetDialogItemText(itemHdl, itemString);
  MoveTo(20, 29);
  DrawString('Search string:  ');
  DrawString(itemString);

  PenNormal;
  SetPort(oldPort);
  end;
    {of procedure DoItemHitModeless}

{ ############################################################################ DoKeyDownModeless }

procedure DoKeyDownModeless(var eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  charCode : char;
  itemType : integer;
  itemHandle : Handle;
  itemRect : Rect;
  finalTicks : UInt32;
  theDialogRef : DialogRef;
  itemHit : integer;
  ignored : boolean;

  begin
  myWindowPtr := FrontWindow;
  charCode := chr(BAnd(eventRec.message, charCodeMask));

  if ((charCode = char(kReturn)) or (charCode = char(kEnter)))then
    begin
    GetDialogItem(DialogRef(myWindowPtr), iSearch, itemType, itemHandle, itemRect);
    HiliteControl(ControlHandle(itemHandle), kControlButtonPart);
    Delay(8, finalTicks);
    HiliteControl(ControlHandle(itemHandle), 0);
    DoItemHitModeless(DialogRef(myWindowPtr));
    end

  else begin
    theDialogRef := DialogRef(myWindowPtr);
    ignored := DialogSelect(eventRec, theDialogRef, itemHit);
```

```
338       end;
339
340     end;
341       {of procedure DoKeyDownModeless}
342
343   { ############################################################################ DoUpdateDocument }
344
345   procedure DoUpdateDocument(var eventRec : EventRecord);
346
347     var
348     myWindowPtr : WindowPtr;
349     paintRect : Rect;
350     fillPattern : Pattern;
351
352     begin
353     myWindowPtr := WindowPtr(eventRec.message);
354
355     BeginUpdate(myWindowPtr);
356
357     if not (EmptyRgn(myWindowPtr^.visRgn)) then
358       begin
359       SetPort(myWindowPtr);
360
361       EraseRgn(myWindowPtr^.visRgn);
362
363       paintRect := myWindowPtr^.portRect;
364       paintRect.right := paintRect.right - 15;
365       paintRect.bottom := paintRect.bottom - 15;
366       GetIndPattern(fillPattern, 0, 16);
367       FillRect(paintRect, fillPattern);
368
369       DrawGrowIcon(myWindowPtr);
370       end;
371
372     EndUpdate(myWindowPtr);
373     end;
374       {of procedure DoUpdateDocument}
375
376   { ################################################################ DoUpdateMovableOrModeless }
377
378   procedure DoUpdateMovableOrModeless(var eventRec : EventRecord);
379
380     var
381     myWindowPtr : WindowPtr;
382
383     begin
384     myWindowPtr := WindowPtr(eventRec.message);
385
386     BeginUpdate(myWindowPtr);
387     UpdateDialog(myWindowPtr, myWindowPtr^.visRgn);
388     EndUpdate(myWindowPtr);
389     end;
390       {of procedure DoUpdateMovableOrModeless}
391
392   { ############################################################################ DoActivateDocument }
393
394   procedure DoActivateDocument(myWindowPtr : WindowPtr; becomingActive : boolean);
395
396     begin
397     if (becomingActive) then
398       DoAdjustMenus;
399
400     DrawGrowIcon(myWindowPtr);
401     end;
402       {of procedure DoActivateDocument}
403
404   { ############################################################## DrawDefaultButtonOutline }
405
406   procedure DrawDefaultButtonOutline(myDialogRef : DialogRef; theItem : integer);
407
408     var
409     oldPort : WindowPtr;
410     oldPenState : PenState;
411     itemType : integer;
412     itemHandle : Handle;
413     itemRect : Rect;
```

```
414     colGrafPtr : CGrafPtr;
415     isColour : boolean;
416     buttonOval : SInt8;
417     backColour : RGBColor;
418     foreSaveColour : RGBColor;
419     newForeColour : RGBColor;
420     newGray : boolean;
421     targetDevice : GDHandle;
422
423     begin
424     GetPort(oldPort);
425     GetPenState(oldPenState);
426
427     GetDialogItem(myDialogRef, iOK, itemType, itemHandle, itemRect);
428     SetPort(ControlHandle(itemHandle)^^.contrlOwner);
429     InsetRect(itemRect, -4, -4);
430
431     colGrafPtr := CGrafPtr(ControlHandle(itemHandle)^^.contrlOwner);
432
433     if (BAnd(BSR(colGrafPtr ^.portVersion, 14), $00000003) <> 0)
434       then isColour := true
435       else isColour := false;
436
437     buttonOval := trunc((itemRect.bottom - itemRect.top) / 2) + 2;
438
439     if (ControlHandle(itemHandle)^^.contrlHilite = 255)
440       then begin
441       newGray := false;
442
443       if (isColour) then
444         begin
445         GetBackColor(backColour);
446         GetForeColor(foreSaveColour);
447         newForeColour := foreSaveColour;
448         targetDevice := GetMainDevice;
449         newGray := GetGray(targetDevice, backColour, newForeColour);
450         end;
451
452       if (newGray)
453         then RGBForeColor(newForeColour)
454         else PenPat(qd.gray);
455
456       PenSize(3, 3);
457       FrameRoundRect(itemRect, buttonOval, buttonOval);
458
459       if (isColour) then
460         RGBForeColor(foreSaveColour);
461       end
462
463     else begin
464       PenPat(qd.black);
465       PenSize(3, 3);
466       FrameRoundRect(itemRect, buttonOval, buttonOval);
467       end;
468
469     SetPenState(oldPenState);
470     SetPort(oldPort);
471     end;
472       {of procedure DrawDefaultButtonOutline}
473
474   { ############################################################## DoActivateMovableModal }
475
476   procedure DoActivateMovableModal(myWindowPtr : WindowPtr; becomingActive : boolean);
477
478     var
479     a, itemType : integer;
480     itemHdl : Handle;
481     itemRect : Rect;
482
483     begin
484     if (becomingActive)
485       then begin
486       for a := iOK to iWaterColour do
487         begin
488         if (a <> iUserItem) then
489           begin
```

```
          GetDialogItem(DialogRef(myWindowPtr), a, itemType, itemHdl, itemRect);
          HiliteControl(ControlHandle(itemHdl), 0);
          end;
        end;
      DrawDefaultButtonOutline(DialogRef(myWindowPtr), iOK);
      DoAdjustMenus;
      end

      else begin
      for a := iOK to iWaterColour do
        begin
        if (a <> iUserItem) then
          begin
          GetDialogItem(DialogRef(myWindowPtr), a, itemType, itemHdl, itemRect);
          HiliteControl(ControlHandle(itemHdl), 255);
          end;
        end;
      DrawDefaultButtonOutline(DialogRef(myWindowPtr), iOK);
      end;
    end;
      {of procedure DoActivateMovableModal}

{ ###################################################################### DoActivateModeless }

procedure DoActivateModeless(myWindowPtr : WindowPtr; becomingActive : boolean);

    var
    itemType : integer;
    itemHdl : Handle;
    itemRect : Rect;

    begin
    if (becomingActive)
      then begin
      GetDialogItem(DialogRef(myWindowPtr), iSearch, itemType, itemHdl, itemRect);
      HiliteControl(ControlHandle(itemHdl), 0);

      DrawDefaultButtonOutline(DialogRef(myWindowPtr), iSearch);
      SelectDialogItemText(DialogRef(myWindowPtr), iEditText, 0, 32767);
      gSleepTime := LMGetCaretTime;
      DoAdjustMenus;
      end

      else begin
      GetDialogItem(DialogRef(myWindowPtr), iSearch, itemType, itemHdl, itemRect);
      HiliteControl(ControlHandle(itemHdl), 255);

      DrawDefaultButtonOutline(DialogRef(myWindowPtr), iSearch);
      SelectDialogItemText(DialogRef(myWindowPtr), iEditText, 0, 0);
      gSleepTime := kMaxLong;
      end;
    end;
      {of procedure DoActivateModeless}

{ ############################################################################## DoActivate }

procedure DoActivate(var eventRec : EventRecord);

    var
    myWindowPtr : WindowPtr;
    dialogType : integer;
    becomingActive : boolean;

    begin
    myWindowPtr := WindowPtr(eventRec.message);
    becomingActive := BAnd(eventRec.modifiers, activeFlag) = activeFlag;

    if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
      begin
      dialogType := WindowPeek(myWindowPtr)^.refCon;

      if (dialogType = kMovableModal)  then
        DoActivateMovableModal(myWindowPtr, becomingActive)
      else if (dialogType = kModeless) then
        DoActivateModeless(myWindowPtr, becomingActive);
      end
```

```
566
567     else if (WindowPeek(myWindowPtr)^.windowKind = userKind) then
568       DoActivateDocument(myWindowPtr, becomingActive);
569
570     end;
571       {of procedure DoActivate}
572
573   { ############################################################################## DoOSEvent }
574
575   procedure DoOSEvent(var eventRec : EventRecord);
576
577     var
578     dialogType : integer;
579     myWindowPtr : WindowPtr;
580
581     begin
582     myWindowPtr := FrontWindow;
583
584     case BAnd(BSR(eventRec.message, 24), $000000FF) of
585
586       suspendResumeMessage:
587         begin
588         gInBackground := BAnd(eventRec.message, resumeFlag) = 0;
589
590         if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
591           begin
592           dialogType := WindowPeek(myWindowPtr)^.refCon;
593
594           if (dialogType = kMovableModal) then
595             DoActivateMovableModal(myWindowPtr, not(gInBackground))
596           else if (dialogType = kModeless) then
597             DoActivateModeless(myWindowPtr, not(gInBackground));
598           end
599
600         else if (WindowPeek(myWindowPtr)^.windowKind = userKind) then
601           DoActivateDocument(myWindowPtr, not(gInBackground));
602         end;
603
604       mouseMovedMessage:
605         begin
606         end;
607
608       end;
609         {of outer case statement}
610     end;
611       {of procedure DoOSEvent}
612
613   { ############################################################################## DoUpdate }
614
615   procedure DoUpdate(var eventRec : EventRecord);
616
617     var
618     myWindowPtr : WindowPtr;
619     dialogType : integer;
620
621     begin
622     myWindowPtr := WindowPtr(eventRec.message);
623
624     if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
625       begin
626       dialogType := WindowPeek(myWindowPtr)^.refCon;
627
628       if ((dialogType = kMovableModal) or (dialogType = kModeless)) then
629         DoUpdateMovableOrModeless(eventRec);
630       end
631
632     else if (WindowPeek(myWindowPtr)^.windowKind = userKind) then
633       DoUpdateDocument(eventRec);
634
635     end;
636       {of procedure DoUpdate}
637
638   { ########################################################################## DoHideModeless }
639
640   procedure DoHideModeless;
641
```

```
642     var
643     myWindowPtr : WindowPtr;
644     dialogType : integer;
645
646     begin
647     myWindowPtr := FrontWindow;
648
649     if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
650       begin
651       dialogType := WindowPeek(myWindowPtr)^.refCon;
652
653       if (dialogType = kModeless) then
654         begin
655         HideWindow(myWindowPtr);
656         InvalRgn(gWindowPtr^.visRgn);
657         gSleepTime := kMaxLong;
658         end;
659       end;
660     end;
661       {of procedure DoHideModeless}
662
663   { ############################################################################## DoEditMenu }
664
665   procedure DoEditMenu(menuItem : integer);
666
667     var
668     myWindowPtr : WindowPtr;
669     dialogType : integer;
670
671     begin
672     myWindowPtr := FrontWindow;
673
674     if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
675       begin
676       dialogType := WindowPeek(myWindowPtr)^.refCon;
677
678       if (dialogType = kModeless) then
679         case (menuItem) of
680
681           iCut:
682             begin
683             DialogCut(DialogRef(myWindowPtr));
684             end;
685
686           iCopy:
687             begin
688             DialogCopy(DialogRef(myWindowPtr));
689             end;
690
691           iPaste:
692             begin
693             DialogPaste(DialogRef(myWindowPtr));
694             end;
695
696           iClear:
697             begin
698             DialogDelete(DialogRef(myWindowPtr));
699             end;
700           end;
701             {of case statement}
702       end;
703     end;
704       {of procedure DoEditMenu}
705
706   { ############################################################################## EventFilter }
707
708   function EventFilter(myDialogRef : DialogRef; eventRec : EventRecord;
709                           var itemHit : integer) : boolean;
710
711     var
712     charCode : char;
713     itemType : integer;
714     itemHandle : Handle;
715     itemRect : Rect;
716     finalTicks : UInt32;
717     handledEvent : boolean;
```

```
718
719     begin
720     handledEvent := false;
721
722     if ((eventRec.what = updateEvt) and (WindowPtr(eventRec.message) <> myDialogRef))
723       then DoUpdate(eventRec)
724       else begin
725         case (eventRec.what) of
726
727           keyDown, autoKey:
728             begin
729             charCode := chr(BAnd(eventRec.message, charCodeMask));
730             if ((charCode = char(kReturn)) or (charCode = char(kEnter))) then
731               begin
732               GetDialogItem(myDialogRef, iOK, itemType, itemHandle, itemRect);
733               HiliteControl(ControlHandle(itemHandle), kControlButtonPart);
734               Delay(8, finalTicks);
735               HiliteControl(ControlHandle(itemHandle), 0);
736               handledEvent := true;
737               itemHit := iOK;
738               end;
739             if ((charCode = char(kEscape)) or ((BAnd(eventRec.modifiers, cmdKey) <> 0)
740                 and (charCode = char(kPeriod)))) then
741               begin
742               GetDialogItem(myDialogRef, iCancel, itemType, itemHandle, itemRect);
743               HiliteControl(ControlHandle(itemHandle), kControlButtonPart);
744               Delay(8, finalTicks);
745               HiliteControl(ControlHandle(itemHandle),0);
746               handledEvent := true;
747               itemHit := iCancel;
748               end;
749
750             {Other keyboard equivalents handled here.}
751             end;
752
753           {Disk-inserted and other events handled here.}
754           end;
755             {of case statement}
756       end;
757
758     EventFilter := handledEvent;
759     end;
760       {of procedure EventFilter}
761
762   { ########################################################################## DoModalDialog }
763
764   function DoModalDialog : boolean;
765
766     var
767     modalDlgPtr : DialogRef;
768     itemType, itemHit : integer;
769     itemHdl : Handle;
770     itemRect : Rect;
771
772     begin
773
774     modalDlgPtr := GetNewDialog(rModal, nil, WindowPtr(-1));
775     if (modalDlgPtr = nil) then
776       begin
777       DoModalDialog := false;
778       Exit(DoModalDialog);
779       end;
780
781     GetDialogItem(modalDlgPtr, iUserItem, itemType, itemHdl, itemRect);
782     SetDialogItem(modalDlgPtr, iUserItem, itemType, Handle(@DrawDefaultButtonOutline),itemRect);
783
784     GetDialogItem(modalDlgPtr, iGridSnap, itemType, itemHdl, itemRect);
785     SetControlValue(ControlHandle(itemHdl), gGridSnap);
786
787     GetDialogItem(modalDlgPtr, iShowGrid, itemType, itemHdl, itemRect);
788     SetControlValue(ControlHandle(itemHdl), gShowGrid);
789
790     GetDialogItem(modalDlgPtr, iShowRulers, itemType, itemHdl, itemRect);
791     SetControlValue(ControlHandle(itemHdl), gShowRule);
792
793     ShowWindow(modalDlgPtr);
```

```
794
795     repeat
796       ModalDialog(NewModalFilterProc(ModalFilterProcPtr(@EventFilter)), itemHit);
797       GetDialogItem(modalDlgPtr, itemHit, itemType, itemHdl, itemRect);
798       if (GetControlValue(ControlHandle(itemHdl)) = 1)
799         then SetControlValue(ControlHandle(itemHdl), 0)
800         else if (GetControlValue(ControlHandle(itemHdl)) = 0) then
801           SetControlValue(ControlHandle(itemHdl), 1);
802
803     until ((itemHit = iOK) or (itemHit = iCancel));
804
805     if (itemHit = iOK) then
806       begin
807       GetDialogItem(modalDlgPtr, iGridSnap, itemType, itemHdl, itemRect);
808       gGridSnap := GetControlValue(ControlHandle(itemHdl));
809
810       GetDialogItem(modalDlgPtr, iShowGrid, itemType, itemHdl, itemRect);
811       gShowGrid := GetControlValue(ControlHandle(itemHdl));
812
813       GetDialogItem(modalDlgPtr, iShowRulers, itemType, itemHdl, itemRect);
814       gShowRule := GetControlValue(ControlHandle(itemHdl));
815       end;
816
817     DisposeDialog(modalDlgPtr);
818
819     DoModalDialog := true;
820     end;
821       {of function DoModalDialog}
822
823   { ################################################################## DoMovableModalDialog }
824
825   function DoMovableModalDialog : boolean;
826
827     var
828     modalDlgPtr : DialogRef;
829     itemType : integer;
830     itemHdl : Handle;
831     itemRect : Rect;
832
833     begin
834     modalDlgPtr := GetNewDialog(rMovable, nil, WindowPtr(-1));
835
836     if (modalDlgPtr = nil) then
837       begin
838       DoMovableModalDialog := false;
839       Exit(DoMovableModalDialog);
840       end;
841
842     SetWRefCon(modalDlgPtr, longint(kMovableModal));
843
844     GetDialogItem(modalDlgPtr, iUserItem, itemType, itemHdl, itemRect);
845
846     SetDialogItem(modalDlgPtr, iUserItem, itemType, Handle(@DrawDefaultButtonOutline),itemRect);
847
848     GetDialogItem(modalDlgPtr, gBrushType, itemType, itemHdl, itemRect);
849     SetControlValue(ControlHandle(itemHdl),1);
850
851     ShowWindow(modalDlgPtr);
852
853     gOldBrushType := gBrushType;
854
855     DoMovableModalDialog := true;
856     end;
857       {of function DoMovableModalDialog}
858
859   { ################################################################### DoModelessDialog }
860
861   function DoModelessDialog : boolean;
862
863     var
864     itemType : integer;
865     itemHdl : Handle;
866     itemRect : Rect;
867
868     begin
869     if (gModelessDlgPtr = nil)
```

```
      then begin
      gModelessDlgPtr := GetNewDialog(rModeless, nil, WindowPtr(-1));
      if (gModelessDlgPtr = nil) then
        begin
        DoModelessDialog := false;
        Exit(DoModelessDialog);
        end;

      SetWRefCon(gModelessDlgPtr, longint(kModeless));

      GetDialogItem(gModelessDlgPtr, iUserItem, itemType, itemHdl, itemRect);
      SetDialogItem(gModelessDlgPtr, iUserItem, itemType, Handle(@DrawDefaultButtonOutline),
                    itemRect);

      ShowWindow(gModelessDlgPtr);
      SelectDialogItemText(gModelessDlgPtr, iEditText, 0, 32767);
      end

      else begin
      ShowWindow(gModelessDlgPtr);
      SelectWindow(gModelessDlgPtr);
      end;

    DoModelessDialog := true;
    end;
      {of function DoModelessDialog}

{ ###################################################################### DoDemonstrationMenu }

procedure DoDemonstrationMenu(menuItem : integer);

    var
    myWindowPtr : WindowPtr;
    theRect : Rect;
    ignored : integer;

    begin
    case (menuItem) of

    iAlert: begin
      myWindowPtr := FrontWindow;
      if (GetAlertStage > 0) then
        begin
        if (myWindowPtr <> nil) then
          begin
          if (WindowPeek(myWindowPtr)^.windowKind <> dialogKind) then
            begin
            SetRect(theRect, myWindowPtr^.portRect.right - 15,
              myWindowPtr^.portRect.bottom - 15, myWindowPtr^.portRect.right,
              myWindowPtr^.portRect.bottom);
            InvalRect(theRect);
            DoActivateDocument(myWindowPtr, false);
            end
          else if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
            begin
            DoActivateModeless(myWindowPtr, false);
            end;
          end;
        end;
      ignored := NoteAlert(rAlert, ModalFilterUPP(@EventFilter));
      end;

    iModal: begin
      myWindowPtr := FrontWindow;
      if (myWindowPtr <> nil) then
        begin
        if (WindowPeek(myWindowPtr)^.windowKind <> dialogKind) then
          begin
          SetRect(theRect, myWindowPtr^.portRect.right - 15,
            myWindowPtr^.portRect.bottom - 15, myWindowPtr^.portRect.right,
            myWindowPtr^.portRect.bottom);
          InvalRect(theRect);
          DoActivateDocument(myWindowPtr, false);
          end
        else if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
          begin
```

```
946           DoActivateModeless(myWindowPtr, false);
947           end;
948         end;
949
950       if not (DoModalDialog) then
951         begin
952         SysBeep(10);
953         ExitToShell;
954         end;
955       end;
956
957     iMovable: begin
958       if not (DoMovableModalDialog) then
959         begin
960         SysBeep(10);
961         ExitToShell;
962         end;
963       end;
964
965     iModeless: begin
966       if not (DoModelessDialog) then
967         begin
968         SysBeep(10);
969         ExitToShell;
970         end;
971       end;
972     end;
973       {of case statement}
974
975     end;
976       {of procedure DoDemonstrationMenu}
977
978   { ############################################################################## DoMenuChoice }
979
980   procedure DoMenuChoice(menuChoice : longint);
981
982     var
983     menuID, menuItem : integer;
984     itemName : string;
985     daDriverRefNum : integer;
986
987     begin
988     menuID := HiWord(menuChoice);
989     menuItem := LoWord(menuChoice);
990
991     if (menuID = 0) then
992       Exit(DoMenuChoice);
993
994     case (menuID) of
995
996       mApple:
997         begin
998         if (menuItem = iAbout)
999           then SysBeep(10)
1000          else begin
1001            GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
1002            daDriverRefNum := OpenDeskAcc(itemName);
1003            end;
1004        end;
1005
1006      mFile:
1007        begin
1008        if (menuItem = iQuit)
1009          then gDone := true
1010          else if (menuItem = iClose) then
1011              DoHideModeless;
1012        end;
1013
1014      mEdit:
1015        begin
1016        DoEditMenu(menuItem);
1017        end;
1018
1019      mDemonstration:
1020        begin
1021        DoDemonstrationMenu(menuItem);
```

```
      end;
    end;
      {of case statement}

  HiliteMenu(0);
  end;
    {of procedure DoMenuChoice}

{ ############################################################################ DoKeyDownDocument }

procedure DoKeyDownDocument(var eventRec : EventRecord);

  var
  charCode : char;

  begin
  charCode := chr(BAnd(eventRec.message, charCodeMask));

  if (BAnd(eventRec.modifiers, cmdKey) <> 0) then
    begin
    DoAdjustMenus;
    DoMenuChoice(MenuKey(charCode));
    end;
  end;
    {of procedure DoKeyDownDocument}

{ ################################################################################ DoKeyDown }

procedure DoKeyDown(var eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  dialogType : integer;

  begin
  myWindowPtr := FrontWindow;

  if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
    begin
    dialogType := WindowPeek(myWindowPtr)^.refCon;

    case (dialogType) of

      kMovableModal:
        begin
        DoKeyDownMovableModal(eventRec);
        end;

      kModeless:
        begin
        DoKeyDownModeless(eventRec);
        end;
      end;
        {of case statement}
    end

  else if (WindowPeek(myWindowPtr)^.windowKind = userKind) then
    DoKeyDownDocument(eventRec);

  end;
    {of procedure DoKeyDown}

{ ################################################################## DoItemHitMovableModal }

procedure DoItemHitMovableModal(myDialogRef : DialogRef; itemHit : integer);

  var
  a, itemType : integer;
  itemHdl : Handle;
  itemRect : Rect;

  begin
  if ((itemHit = iCharcoal) or (itemHit = iOilPaint) or (itemHit = iWaterColour))
    then begin
    for a := iCharcoal to iWaterColour do
      begin
```

```
1098        GetDialogItem(myDialogRef, a, itemType, itemHdl, itemRect);
1099        SetControlValue(ControlHandle(itemHdl), 0);
1100        end;
1101
1102      GetDialogItem(myDialogRef, itemHit, itemType, itemHdl, itemRect);
1103      SetControlValue(ControlHandle(itemHdl), 1);
1104      gBrushType := itemHit;
1105      end
1106
1107    else begin
1108      if ((itemHit = iOK) or (itemHit = iCancel)) then
1109        begin
1110        if (itemHit = iCancel) then
1111          gBrushType := gOldBrushType;
1112        DisposeDialog(myDialogRef);
1113        end;
1114      end;
1115    end;
1116      {of procedure DoItemHitMovableModal}
1117
1118  { ############################################################## InvalidateScrollBarArea }
1119
1120  procedure InvalidateScrollBarArea(myWindowPtr : WindowPtr);
1121
1122    var
1123    tempRect : Rect;
1124
1125    begin
1126    SetPort(myWindowPtr);
1127
1128    tempRect := myWindowPtr^.portRect;
1129    tempRect.left := tempRect.right - 15;
1130    InvalRect(tempRect);
1131
1132    tempRect := myWindowPtr^.portRect;
1133    tempRect.top := tempRect.bottom - 15;
1134    InvalRect(tempRect);
1135    end;
1136      {of procedure InvalidateScrollBarArea}
1137
1138  { ########################################################################## DoInContent }
1139
1140  procedure DoInContent(var eventRec : EventRecord);
1141
1142    var
1143    myWindowPtr : WindowPtr;
1144    dialogType : integer;
1145    myDialogRef : DialogRef;
1146    itemHit : integer;
1147
1148    begin
1149    myWindowPtr := FrontWindow;
1150
1151    if (WindowPeek(myWindowPtr)^.windowKind = dialogKind) then
1152      begin
1153      dialogType := WindowPeek(myWindowPtr)^.refCon;
1154
1155      if (dialogType = kMovableModal) then
1156        begin
1157        if (DialogSelect(eventRec, myDialogRef, itemHit)) then
1158          DoItemHitMovableModal(myDialogRef, itemHit);
1159        end
1160
1161      else if (dialogType = kModeless) then
1162        begin
1163        if (DialogSelect(eventRec, myDialogRef, itemHit)) then
1164          DoItemHitModeless(myDialogRef);
1165        end;
1166      end
1167
1168    else if (WindowPeek(myWindowPtr)^.windowKind = userKind) then
1169      begin
1170      { Handle clicks in document content region here.}
1171      end;
1172    end;
1173      {of procedure DoInContent}
```

```
1174
1175  { ############################################################################## DoMouseDown }
1176
1177  procedure DoMouseDown(eventRec : EventRecord);
1178
1179    var
1180    myWindowPtr : WindowPtr;
1181    partCode : integer;
1182    growRect : Rect;
1183    newSize : longint;
1184
1185    begin
1186    partCode := FindWindow(eventRec.where, myWindowPtr);
1187
1188    case (partCode) of
1189
1190      inMenuBar:
1191        begin
1192        DoAdjustMenus;
1193        DoMenuChoice(MenuSelect(eventRec.where));
1194        end;
1195
1196      inSysWindow:
1197        begin
1198        SystemClick(eventRec, myWindowPtr);
1199        end;
1200
1201      inContent:
1202        begin
1203        if (myWindowPtr <> FrontWindow)
1204          then begin
1205          if (WindowPeek(FrontWindow)^.refCon = kMovableModal)
1206            then SysBeep(10)
1207            else SelectWindow(myWindowPtr);
1208          end
1209
1210          else DoInContent(eventRec);
1211        end;
1212
1213      inDrag:
1214        begin
1215        if ((WindowPeek(FrontWindow)^.refCon = kMovableModal) and
1216           (WindowPeek(myWindowPtr)^.refCon <> kMovableModal)) then
1217          begin
1218          SysBeep(10);
1219          Exit(DoMouseDown);
1220          end;
1221        DragWindow(myWindowPtr, eventRec.where, qd.screenBits.bounds);
1222        end;
1223
1224      inGoAway:
1225        begin
1226        if (TrackGoAway(myWindowPtr, eventRec.where)) then
1227          DoHideModeless;
1228        end;
1229
1230      inGrow:
1231        begin
1232        growRect := qd.screenBits.bounds;
1233        growRect.top := 80;
1234        growRect.left := 160;
1235        newSize := GrowWindow(myWindowPtr, eventRec.where, growRect);
1236        if (newSize <> 0) then
1237          begin
1238          InvalidateScrollBarArea(myWindowPtr);
1239          SizeWindow(myWindowPtr, LoWord(newSize), HiWord(newSize), true);
1240          InvalidateScrollBarArea(myWindowPtr);
1241          end;
1242        end;
1243
1244      end;
1245        {of case statement}
1246    end;
1247      {of procedure DoMouseDown}
1248
1249  { ################################################################################ DoEvents }
```

```
1250
1251  procedure DoEvents(eventRec : EventRecord);
1252
1253    begin
1254    case (eventRec.what) of
1255
1256      mouseDown:
1257        begin
1258        DoMouseDown(eventRec);
1259        end;
1260
1261      keyDown, autoKey:
1262        begin
1263        DoKeyDown(eventRec);
1264        end;
1265
1266      updateEvt:
1267        begin
1268        DoUpdate(eventRec);
1269        end;
1270
1271      activateEvt:
1272        begin
1273        DoActivate(eventRec);
1274        end;
1275
1276      osEvt:
1277        begin
1278        DoOSEvent(eventRec);
1279        HiliteMenu(0);
1280        end;
1281      end;
1282        {of case statement}
1283    end;
1284      {of procedure DoEvents}
1285
1286  { ############################################################################## EventLoop }
1287
1288  procedure EventLoop;
1289
1290    var
1291    eventRec : EventRecord;
1292    gotEvent : Boolean;
1293
1294    begin
1295    gSleepTime := kMaxLong;
1296
1297    gDone := false;
1298
1299    while not (gDone) do
1300      begin
1301      gotEvent := WaitNextEvent(everyEvent, eventRec, gSleepTime, nil);
1302
1303      if (gotEvent)
1304        then DoEvents(eventRec)
1305        else DoIdle(eventRec);
1306      end;
1307    end;
1308      {of procedure EventLoop}
1309
1310
1311  { ################################################################## start of main program }
1312
1313  begin
1314
1315    gGridSnap := 0;
1316    gShowGrid := 0;
1317    gShowRule := 0;
1318    gBrushType := iCharcoal;
1319    gOldBrushType := iCharcoal;
1320    gModelessDlgPtr := nil;
1321
1322    { ……… initialize managers }
1323
1324    DoInitManagers;
1325
```

```
1326    { ……… set up menu bar and menus }
1327
1328    menubarHdl := GetNewMBar(rMenubar);
1329    if (menubarHdl = nil) then
1330      ExitToShell;
1331    SetMenuBar(menubarHdl);
1332    DrawMenuBar;
1333
1334    menuHdl := GetMenuHandle(mApple);
1335    if (menuHdl = nil)
1336      then ExitToShell
1337      else AppendResMenu(menuHdl,'DRVR');
1338
1339    { ……… open window }
1340
1341    gWindowPtr := GetNewWindow(rNewWindow, nil, WindowPtr(-1));
1342    if (gWindowPtr = nil) then
1343      ExitToShell;
1344
1345    docRecHdl := DocRecHandle(NewHandle(sizeof(DocRec)));
1346    if (docRecHdl = nil) then
1347      ExitToShell;
1348
1349    SetWRefCon(gWindowPtr, longint(docRecHdl));
1350
1351    { ……… enter eventLoop }
1352
1353    EventLoop;
1354
1355  end.
1356
1357  { ############################################################################### }
```

Demonstration Program Comments

When this program is run, the user should:

- Invoke alerts and dialog boxes by selecting items in the Demonstration menu, noting window update/activation/deactivation and menu enabling/disabling effects.
- Note particularly the effects on the Apple, Help, and Application menus when alert, modal, movable modal and modeless dialog boxes are the frontmost window.
- Click outside the alert box and modal dialog box when they are the frontmost window, noting that the only response is the system alert sound.
- Note that, when the movable modal dialog box is displayed:
    - The alert sound is played when the user clicks in both the window's content region and its title bar.
    - The program can be sent to the background by clicking outside the dialog box and window or by selecting another application from the Application menu.
    - The program can be brought to the foreground again by clicking inside the dialog box or application window or by selecting the program from the Application menu.
- Note that, when the modeless dialog box is displayed:
    - It behaves like a normal document window when the user:
        - Clicks outside it (or selects another application from the Application menu) when it is the frontmost window.
        - Clicks inside it (or selects the application from the Application menu) when it is not the frontmost window.
    - It can be hidden by clicking in the close box or by selecting Close from the File menu.
    - An alert, modal dialog box or movable modal dialog box can be invoked "on top of" the modeless dialog box.

    - The Edit menu Cut, Copy, Paste and Clear commands are enabled and support editing in the editable text item.

- Note that the movable modal and modeless dialog boxes respond correctly to the Return, Enter and Esc keys, and to the Command-period keyboard combination.

- Note that the 'ALRT' resource is defined to play the alert sound only at the first invocation of the alert, display the alert box and play the alert sound once at the second invocation, display the alert box and play the alert sound twice at the third invocation, and display the alert box and play the alert sound three times at the fourth and subsequent invocations.

- Note that, when the movable modal dialog and modeless dialog boxes are not the frontmost window, the default button bold outline is dimmed.

- Select Show Balloons from the Help menu while an alert box or dialog box is the frontmost window, cause balloons to open over the boxes and note the updating of the box behind the balloon when the balloon closes. Note that the system does not redraw the icon or the bold outline of the default button of an alert box after it has been obscured.

The constant declaration block

Lines 61-92 establish constants relating to menu and window resources, alert box and dialog boxes resources and item numbers, menu IDs and menu item numbers. Lines 94-95 establish constants which will be assigned to the refCon field of the window records associated with the movable modal dialog box and the modeless dialog box. Lines 98-101 establish constants representing the character codes for the Return, Enter, Esc, and period keys.

Line 102 defines kMaxLong as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

The type declaration block

At Lines 108-113, a data type for a document record is created. The elements of the document record will not actually be used in this demonstration. The document record handle will simply be assigned to the refCon field of the normal window’s window record.

The variable declaration block

gWindowPtr will be assigned the pointer to the single window opened by the program. gSleepTime will be assigned the value which will be used as the sleep parameter in the WaitNextEvent call. (This value will be changed during program execution.) gDone controls the exit from the main event loop. gInBackground relates to foreground/background switching.

The global variables at Lines 123-125 will contain the current setting of the checkboxes in the modal dialog box. The global variables at Lines 126-127 will contain the identity of the newly selected and previously selected radio buttons in the movable modal dialog box.

Line 128 declares the pointer to the dialog record for the modeless dialog box as a global variable because, when the dialog is invoked by the user, the program needs to know whether the dialog has never been opened or whether it has previously been opened but is currently hidden.

The procedure DoIdle

DoIdle is invoked whenever WaitNextEvent returns a null event.

Line 165 gets a pointer to the front window. If the window is one of the dialog windows (Line 166), Line 168 retrieves the dialog type from the window record's refCon. If the window is the modeless dialog (which contains an editable text item), DialogSelect is called (Lines 170-171). DialogSelect, amongst other things, calls TEIdle, which blinks the insertion point caret. (As will be seen, WaitNextEvent's sleep parameter is changed from kMaxLong whenever the modeless dialog box is the frontmost window, thus causing null events to be received at a rate equal to the currently set caret blink rate.)

The procedure DoKeyDownMovableModal

DoKeyDownMovableModal continues key-down processing for key-downs in the movable modal dialog box.

If the character code of the key equals the character code returned by the Return or Enter keys (Line 252), the handle to the control record for the OK button control is obtained by the

call to GetDialogItem (Line 254) and used at Lines 255-257 to highlight the OK button for 8 ticks. The dialog box is then closed down (Line 258).

If the character code of the key equals the character code returned by the Esc key, or if the Command and period keys were both down (Lines 261-262), the handle to the control record for the Cancel button control is obtained by the call to GetDialogItem (Line 264) and used at Lines 265-267 to highlight the Cancel button for 8 ticks. Before the dialog box is closed down (Line 269), and since the user has clicked the Cancel button, the value in the global variable which keeps track of the currently selected radio button is made equal to the value that was assigned to that variable before the dialog was invoked (Line 268).

The procedure DoItemHitModeless

DoItemHitModeless further processes, to completion, a mouse-down event in an enabled control in the modeless dialog box.

Since the modeless dialog box has only one control (the Search (OK) button), the item hit must have been that button. Accordingly, Line 295 gets a handle to the editable text item, which is used at Line 296 to retrieve the string in the editable text item.

The rest of the code is concerned only with printing the retrieved text string in the window.

The procedure DoKeyDownModeless

DoKeyDownModeless continues key-down processing for key-downs in the modeless dialog box.

This procedure performs the same button highlighting in response to Return and Enter key-downs as did the previous procedure. (The modeless dialog box has only one button — the Start (OK) button.) Note, however, that the dialog box is not dismissed after a Return or Enter key is pressed. Instead, the application-defined procedure DoItemHitModeless is called (Line 331). As will be seen, that procedure extracts the text string from the editable text item.

If, however, the event did not arise from a Return or Enter key press (Line 334), the focus changes to the editable text item. Accordingly, at Line 336, DialogSelect is called to handle the event automatically in conjunction with TextEdit, the visual result being the appearance of the character in the editable text item display.

The procedure DoUpdateDocument

DoUpdateDocument simply fills the content region (less the scroll bar areas) of the window with one of the system patterns to assist in visually "proving" correct window updating.

The procedure DoUpdateMovableOrModeless

The update task for both movable modal and modeless dialog boxes is the same, that is, redraw the update region. Accordingly, the procedure DoUpdateMovableOrModeless calls UpdateDialog between calls to BeginUpdate and EndUpdate (Lines 385-387) to achieve this.

The procedure DoActivateDocument

DoActivateDocument performs window activation for the document window. If the window is becoming active, the menus are adjusted as appropriate for a document window (Line 396-397). Regardless of whether the window is being activated or deactivated, DrawGrowIcon is called (Line 399). (DrawGrowIcon "knows" whether the window is becoming active or inactive and draws the grow icon or an empty size box accordingly.)

The procedure DrawDefaultButtonOutline

DrawDefaultButtonOutline is the application-defined function for drawing the bold outline around the default button in the modal, movable modal and modeless dialog boxes. Recall that, in DoModalDialog, DoMovableModalDialog, and DoModelessDialog, a pointer to this draw function was installed in the user item in the modal, movable modal, and modeless dialog boxes. The consequence of that is that this function will be called whenever the user item is part of the dialog box's update region during a dialog box update.

Firstly, a pointer to the current graphics port is saved, as is the current pen state (Lines 423-424).

A handle to the OK button's control record, together with the button's display rectangle, is retrieved at Line 426. The handle is used at Line 427 to retrieve the pointer to the control's owner window from the contrlOwner field of the control record. The SetPort call at Line 427 uses this window pointer to set the current graphics port. The InsetRect call at

Line 428 expands the returned rectangle by 4 pixels top and bottom and left and right, that is, to the desired outside boundaries of the bold outline.

The next step is to determine whether the dialog box is using a colour graphics port. The following is relevant to this step:

- The seventh and eighth bytes in a colour graphics port constitute the portVersion field. The two high bits of this word are invariably set.

- The seventh and eighth bytes in a non-colour graphics port constitute the rowBytes field of the port's portBits field. The high two bits of the rowBytes field are invariably clear.

At Line 430, the window pointer in the contrlOwner field of the OK button's control record is cast to a pointer to a colour graphics port so that the two top bits can be examined as if they are part of the portVersion field of a colour graphics port. (Of course, if we are dealing with a non-colour graphics port, the bits will actually be the two top bits of the rowBytes field.) At Line 432, the bits are tested. If the test indicates that the bits are set, the port must be a colour graphics port, in which case the variable isColour is set to true, otherwise it is set to false (Lines 433-434).

At Line 436, the variable which will control the curvature of the corners of the bold outline is set to the appropriate value based on the vertical dimension of the OK box's display rectangle.

The bold outline must be drawn in black if the OK button is active and in gray (that is, either a gray colour or the gray pattern) if it is inactive. Accordingly, Line 438 examines the controlHilite field of the OK button's control record to determine whether the control is currently inactive or active.

Lines 440-459 deal with the case of an inactive OK button. Firstly, newGray is set to false (Line 440) preparatory to possible modification in the next eight lines of code.

If the dialog is using a colour graphics port (Line 442), the current background and foreground colours are assigned to two RGBColor variables (Lines 444-445) and the variable newForeColour is made equal to the foreground colour (Line 446). Line 447 retrieves a handle to the main graphics device, that is, to the screen which carries the menu bar. This handle is required by the call to GetGray at Line 448. GetGray provides the best available gray between the two colours passed in the second and third parameters for the device specified in the first parameter. GetGray returns true if at least one gray or intermediate colour is available, in which case the third parameter will contain that gray or intermediate colour.

If GetGray was successful, the colour returned is used to set the foreground drawing colour (Line 452). Otherwise, the current pen pattern is set to gray (Line 453). (Note that gray is a QuickDraw global variable specifying a pattern, not a colour.)

Having determined whether to draw the bold outline as a gray colour or as a gray pattern, the next step is to draw the outline. Accordingly, Lines 455-456 set the pen size and draw the round-cornered rectangle with a call to FrameRoundRect. It then remains to restore the foreground colour to its previous value, if necessary (Lines 458-459).

If the test at Line 439 revealed that the OK button was active, Lines 463-465 simply set the pen pattern to black and draw the round-cornered rectangle.

The function restores the old pen state and graphics port before returning (Lines 468-469).

The procedure DoActivateMovableModal

DoActivateMovableModal performs window activation and deactivation for the movable modal dialog box.

If the dialog box is becoming active, a handle to each of the dialog's control records is obtained with GetDialogItem, and HiliteControl is called to make the associated controls active and undimmed (Lines 483-492). In addition, an application-defined procedure which draws the bold outline around the default button is called (Line 493) and the menus are adjusted as appropriate for the movable modal dialog (Line 494).

If the dialog box is becoming inactive, the controls are made inactive and dimmed (497-505) and the outline around the default button is drawn in gray (Line 506).

The procedure DoActivateModeless

DoActivateModeless performs window activation and deactivation for the modeless dialog box.

If the dialog box is becoming active (Line 521), its control is made active and undimmed (Lines 523-524), and the bold outline around the single button is drawn in black (Line 526). The call to SelectDialogItemText at Line 527 causes the insertion point caret to blink (if there is no text in the item) or the text to be selected (if there is text in the item). Line 528 sets the variable used in the sleep parameter in the WaitNextEvent call to equal the value returned by LMGetCaretTime (which is the value set by the user at the Insertion Point Blinking section in the General Controls control panel). Line 529 adjusts the menus as appropriate for the modeless dialog box.

If the dialog box is becoming inactive (Line 532), its control is made inactive and dimmed (Lines 533-534), the bold outline around the default button is drawn in gray (Line 536), selected text is de-selected (Line 537) and the variable used in the sleep parameter of the WaitNextEvent call is reset to kMaxLong (Line 538).

The procedure DoActivate

DoActivate performs initial processing of activate events. If the window is a dialog window (Line 556), and if it is either the movable modal or modeless dialog, the appropriate application-defined activation procedure is called (Lines 560-563). However, if the window is the normal window, the application-defined procedure DoActivateDocument is called (Lines 566-567).

The procedure DoOSEvent

DoOSEvent handles operating system events, branching according to whether the event is a suspend/resume event or a mouse-moved event (Line 583). If the event is a suspend/resume event (Line 585), DoOSEvent calls the appropriate window activation procedure depending on whether the window is the movable modal dialog, the modeless dialog, or the normal window (Lines 589-600), indicating to that function whether to activate or deactivate the window.

The procedure DoUpdate

DoUpdate performs initial processing of update events. If the window is one of the dialog windows (Line 623), and if it is either the movable modal or modeless dialog (Line 627), the application-defined procedure DoUpdateMovableOrModeless is called (Line 628). If, however, the window is the normal window, the application-defined procedure DoUpdateDocument is called (Lines 631-632).

The procedure DoHideModeless

DoHideModeless hides the modeless dialog box. Line 646 gets a pointer to the front window. If the front window is a dialog (Line 648), and if it is the modeless dialog (Lines 652), HideWindow is called at Line 654 to deactivate the dialog box, make it invisible, and activate the window immediately behind. In addition, and since caret blinking in the editable text item is no longer required, the variable which determines the sleep parameter in the WaitNextEvent call is set back to kMaxLong (Line 656).

The InvalRgn call at Line 655 is included simply to force a redraw of the window, thus erasing the text string drawn in the window if the dialog's Search (OK) button was clicked during execution of the DoItemHitModeless function.

The procedure DoEditMenu

DoEditMenu first determines whether the front window is the modeless dialog (Lines 671-673). In this program, the Edit menu is only enabled when the modeless dialog box is the frontmost window. Accordingly, if the front window is the modeless dialog, Cut, Copy, Paste, and Clear selections from the Edit menu will cause the appropriate TextEdit routines to be called to perform those operations on selected text in the editable text item (LInes 677-699).

The function EventFilter

EventFilter is the application-defined event filter function which, in conjuction with ModalDialog, handles events in the alert box and the modal dialog box. In this program, a ProcPtr to EventFilter is passed as the first parameter in the NoteAlert and ModalDialog calls. Note that EventFilter's fourth parameter is a variable parameter.

The application-defined event filter function is necessary to compensate for certain inadequacies of the standard event filter function. It is required to return true if it handled the event or false if it wants the Dialog Manager to process the event. Line 719 sets the variable which will be used to return true or false to ModalDialog and NoteAlert to an initial value of false.

If the event is an update event not belonging to the alert box or modal dialog box (Line 721), the application-defined function DoUpdate is called to update the window specified in the message field of the event record (Line 722). In this program, that window could be either the window or the modeless dialog box. Note also that, by responding to update events in your own inactive windows in this way, you allow ModalDialog to perform a minor switch when necessay so that background applications can update their windows as well. (It may be of interest to remove the DoUpdate call and observe the effect of Help balloons on the application's window and on windows belonging to other applications.)

If the event is a key-down or autokey event (Line 726), and if the character code is that for the Return key or the Enter key (Line 729), Lines 731-736 highlight the OK button for eight ticks, assign true to the variable which contains the function's return value, and assign the item number of the OK button to the itemHit variable. (The value in this variable will be returned by ModalDialog.)

If the event is a key-down or autokey event (Line 726), and if the character code and an examination of the event record modifiers field indicates that either the Esc key or the Command-period combination was pressed (Lines 738-739), Lines 740-746 highlight the Cancel button for eight ticks, assign true to the variable which contains the function's return value and assigns the item number of the Cancel button to the itemHit variable. (The value in this variable will be returned by ModalDialog.)

At Line 757, true is returned if the event was a key-down or autokey event related to the OK or Cancel buttons, causing ModalDialog to ignore these events. Otherwise false is returned, indicating that ModalDialog should process the event itself. (The effect of returning false from this event filter in this program is that ModalDialog will handle all mouse events in the alert box or modal dialog box and all update events related to the alert or dialog box only.)

The function DoModalDialog

DoModalDialog creates, manages and disposes of the modal dialog.

At Line 773, the call to GetNewDialog creates the dialog from the specified resource as the frontmost window.

The GetDialogItem call (Line 780) specifies this dialog's user item number at the second parameter and will thus return, in the fourth parameter, the address at which to install the pointer to the application-defined draw function for drawing the bold outline around the default button. (The user item display rectangle overlays the default button display rectangle.) The SetDialogItem call at Line 781 installs the draw function.

Lines 783-790 obtain handles to the three checkbox controls for the purposes of setting the value of these controls to the values contained in the global variables relating to each control. With the dialog fully prepared, it is made visible by the call to ShowWindow at Line 792.

The repeat/until loop at Lines 794-802 continues to call ModalDialog until the itemHit variable signifies that the OK or Cancel button has been "hit". Note that the first parameter in the ModalDialog call is a pointer to the application-defined event filter function. The second parameter receives the item number of the "hit" item. ModalDialog retains control until one of the checkboxes or one of the buttons is "hit". If a checkbox is clicked, the handle to the item is retrieved for the purposes of flipping the relevant checkbox's value (Lines 797-800) and the loop continues.

When the loop exits, and if the user "hit" the OK button (Line 804), handles to each of the three checkboxes are retrieved for the purposes of retrieving the control's value and assigning it to the relevant global variable. (If the user "hit" the Cancel button, the global variables retain the values they contained before the dialog was displayed.)
The dialog is then disposed of (Line 816).

The function DoMovableModalDialog

DoMovableModalDialog creates the movable modal dialog.

The call to GetNewDialog at Line 833 creates the dialog and the call to SetWRefCon at Line 841 assigns the constant kMovableModal to the refCon field of the window record associated with the dialog. The application-defined function for drawing the bold outline around the default button is installed at Lines 843-845.

At Lines 847-848, the current radio button item number stored in the global variable gBrushType is used to retrieve a handle to the item, which is then used in the SetControlValue call to set that particular button. With the dialog fully prepared, the call to ShowWindow at Line 850 displays the dialog.

User interaction is handled by the main event loop. Before that interaction begins, the current value in gBrushType is assigned to the global variable gOldBrushType (Line 852). As will be seen, this value will be re-assigned to gBrushType if the user "hits" the dialog's Cancel button.

The function DoModelessDialog

In this program, the modeless dialog is only created once, that is, when the user first selects Modeless… from the Demonstration menu. Clicks in its close box, or selecting Close from the File menu while the modeless dialog is the frontmost window, will cause the dialog box to be hidden, not disposed of.

Accordingly, Line 868 of the DoModelessDialog function first determines whether the modeless dialog box is already open. If it is not, Line 870 creates the modeless dialog, the call to SetWRefCon at Line 877 assigns the constant kModeless to the refCon field of the window record associated with the dialog, Lines 879-881 install the application-defined function for drawing the bold outline around the default button, Line 883 displays the window, and the call to SeIectDialogItemText at Line 884 selects the text in the editable text item item (item contains text) or displays the insertion point (item does not contain text).

If the modeless dialog box has already been opened (Line 887), Lines 888-889 show the hidden dialog box and call SelectWindow to generate the necessary activate events.

User interaction with the modeless dialog box is handled by the main event loop.

The procedure DoDemonstrationMenu

doDemonstrationMenu handles selections from the Demonstration menu, switching according to the menu item passed to it.

If the user chose Alert (Line 908), NoteAlert is called (Line 928). Before calling NoteAlert, however, an application must explicitly deactivate the front document window, if one exists. (In this demonstration, the only document window deactivation action required is to erase the grow icon.) In addition, if a modeless dialog is open and showing, that dialog must also be deactivated.

The 'ALRT' resource specifies that, at the first invocation of the alert, the alert sound is to be played but the alert box itself is not to be displayed. Accordingly, Line 910 ensures that Lines 911-927 will only execute if this is not the first invocation.

If there is at least one window of any type open, and if the front window is not the modeless dialog window (Lines 912-914), Lines 916-919 invalidate the grow icon area so as to force an update event for the window. Line 920 then, in effect, calls DrawGrowIcon to erase the grow box. If, however, there is at least one window of any type open and the front window is the modeless dialog (Line 922), the modeless dialog is deactivated (Line 924).

NoteAlert is called at Line 928. Note that this program uses an application-defined filter function, the address of which is passed as the second parameter in the NoteAlert call. NoteAlert exits when the user clicks one of the buttons or presses the Return, Enter, Esc, or Command-period keys.

If the user chose Modal… (Line 931), the same general procedure is followed except that the call to GetAlrtStage is not made and the the application-defined function for creating, managing and disposing of the modal dialog is called (Lines 932-953). (As will be seen, the application-defined filter function is also used to handle events in the modal dialog box.)

If the user chose Movable Modal…, the application-defined function for creating the movable modal dialog is called (Lines 956-962). (From then on, all events pertaining to the movable modal dialog are handled in the main event loop.)

If the user chose Modeless…, the application-defined function for creating the modeless dialog is called (Lines 964-970). (From then on, all events pertaining to the modeless dialog are handled in the main event loop.)

The procedure DoMenuChoice

DoMenuChoice extracts the menu ID and item ID from the long value passed to it (Lines 987-988) and branches according to the menu ID (provided that the menuID value is not 0, meaning that no item was selected).

If the choice was the Quit item in the File menu menu, gDone is set to true, thus terminating the program (Lines 1007-1008). If the choice was the Close item in the File menu, an application-defined procedure which hides the modeless dialog box is called (Lines 1009-1010).

(In this program, the Close item is only enabled when the modeless dialog box is the front window.)

The procedure DoKeyDownDocument

DoKeyDownDocument continues key-down processing for key-downs in the window. The character code for the key is extracted from the event record (Line 1037). If the Command key was down at the same time (Line 1039), the menus are adjusted and the results of a call to MenuKey are passed to the application-defined function DoMenuChoice (Lines 1040-1043).

The procedure DoKeyDown

DoKeyDown takes the key-down and auto-key events and switches according to the type of window in which the event occurred.

The procedure DoItemHitMovableModal

DoItemHitMovableModal further processes, to completion, a mouse-down event in an enabled control in the movable modal dialog box.

Line 1093 determines whether the mouse-down was in one of the three radio buttons. If so, Lines 1094-1109 reset the control value of all three radio buttons to 0 and Lines 1101-1102 set the control value of the radio button that was clicked to 1. In addition, the global variable which holds the currently set radio button is assigned the item number of the radio button that was clicked (Line 1103).

If the radio buttons were not clicked, Lines 1106-1113 cover the remaining possibilities, that is, a click in either the OK button or the Cancel button. If the Cancel button was clicked, the global variable gBrushType is assigned the value it contained before the session of user interaction with the dialog began (Lines 1109-1110). If either the OK or the Cancel button was clicked, the dialog box is disposed of (Line 1111).

The procedure InvalidateScrollBarAreas

InvalidateScrollBarAreas invalidates the scroll bar areas of the window as part of the usual window management procedures.

The procedure DoInContent

DoInContent continues the content region mouse-down handling initiated by DoMouseDown. DoInContent is called by DoMouseDown only if the mouse-down occured in the frontmost (active) window.

Line 1148 gets a pointer to the frontmost window.

If the frontmost window is a dialog (Line 1150), and if it is the movable modal dialog box (Lines 1152-1154), DialogSelect is called (Line 1156). DialogSelect returns true if the mouse-down occurs in an enabled item, in which case the third parameter contains the item number involved. Thus, if the mouse-down occurred in an enabled item, the application-defined function DoItemHitMovableModal is called (Line 1157) to further process the mouse-down event. (Note that DialogSelect tracks user action after the mouse-button goes down and returns true only if the cursor is still within the control when the mouse button is released.)

If the frontmost window is the modeless dialog box (LIne 1160) and the mouse-down occurred in an enabled item, the application-defined function DoItemHitModeless is called to further process the mouse-down event.

The procedure DoMouseDown

DoMouseDown handles mouse-down events. Mouse-downs in the content region, in the title bar, and in the close box are of significance to the demonstration.

In the event of a mouse-down in the content region (Line 1200), Line 1202 establishes whether the click was in the frontmost window or another window. If the click was not in the frontmost window, and if the front window is the movable modal dialog box, the system alert sound is played (Lines 1204-1205) and the dialog box is retained as the frontmost window. (This action is necessary to preserve the required modal characteristic of movable modal dialog boxes.) If the front window was not the movable modal dialog box, SelectWindow is called (Line 1206) to generate the necessary activate events.

If the mouse-down was in the frontmost window, the application-defined function DoInContent is called to further process the event (Line 1209).

A movable modal dialog box must also remain the frontmost window if the user clicks in the title bar of the application's window. Accordingly, before DragWindow is called to handle a title bar mouse-down (Line 1220), Line 1214 checks to see if the front window is the movable modal dialog box. If it is, and if the event relates to another window (Line 1215), the system alert sound is played and the function returns without calling DragWindow (1217-1218).

If a mouse-down occurs in the close box, and if TrackGoAway returns true (Lines 1223-1225), the application-defined function DoHideModeless is called. (In this demonstration, the modeless dialog box, but not the window, has a close box.)

Lines 1229-1240 provide the usual responses for a mouse-down in the size box of the window.

The procedure DoEvents

DoEvents switches according to the event type reported. (It is important to remember at this point that events which occur when an alert box or modal dialog box has been invoked are not handled by the main event loop and associated event-handling functions.)

The procedure EventLoop

The main event loop continues until gDone is set to true by the user selecting Quit from the File menu.

At Line 1294, the variable which will be used as WaitNextEvent's sleep parameter is set to kMaxLong, indicating that the application has no need for null events and that it will yield the microprocessor to other applications for the maximum possible time if no events are pending for it. Note that the value assigned to gSleepTime will be changed later on, causing null events to be received; hence the call to the idle processing function at Line 1304.

The main program block

The main function initialises the system software managers (Line 1323), sets up the menu bar and menus (Lines 1327-1336), opens a window (Line 1340), creates a relocatable block for the window's window record and assigns the handle to the window record's refCon field (1344-1348), and enters the main event loop (Line 1352).

Note that error handling here and in other areas of the program is somewhat rudimentary. The program simply terminates.

AN ALTERNATIVE APPROACH FOR THE MODAL DIALOG

The following details an alternative approach to achieving keystroke aliasing for the OK and Cancel buttons, and default button outlining, in the modal dialog. This approach involves the use of two Dialog Manager routines (SetDialogDefaultItem and SetDialogCancelItem) for which documentation remains somewhat obscure.

SetDialogTracksCursor is a sister routine introduced with SetDialogDefaultItem and SetDialogCancelItem. If a modal dialog includes one or more editable text items, this routine may be used to automatically change the cursor to the I-beam shape whenever it is over an editable text item. The following also includes a demonstration of the use of this sister routine.

Step 1 is to open the modal dialog's 'DITL' resource, remove the User Item and add an editable text item, taking care not to change the item numbers of the OK and Cancel buttons. (Note that, when the Dialog Manager "sees" the editable text it the item list, it will automatically activate and deactivate the Edit menu and the Cut, Copy, and Paste items when the dialog is opened and closed.)

Step 2 is to replace the DoModalDialog function with the following version:

```
function DoModalDialog : boolean;

  var
  modalDlgPtr : DialogPtr;
  itemType, itemHit : integer;
  itemHdl : Handle;
  itemRect : Rect;
  osError : OSErr;

  begin
  modalDlgPtr := GetNewDialog(rModal, nil, WindowPtr(-1));
  if(modalDlgPtr = nil) then
```

```
    begin
    DoModalDialog := false;
    Exit(DoModalDialog);
    end;

  { Installation of DrawDefaultButtonOutline function removed from here. }

  GetDialogItem(modalDlgPtr, iGridSnap, itemType, itemHdl, itemRect);
  SetControlValue(ControlHandle(itemHdl), gGridSnap);

  GetDialogItem(modalDlgPtr, iShowGrid, itemType, itemHdl, itemRect);
  SetControlValue(ControlHandle(itemHdl), gShowGrid);

  GetDialogItem(modalDlgPtr, iShowRulers, itemType, itemHdl, itemRect);
  SetControlValue(ControlHandle(itemHdl), gShowRule);

  { SetDialogDefaultItem will enable automatic keyboard aliasing for the OK button and
    will also cause a bold outline to be drawn around that button.  SetDialogCancelItem
    will enable automatic keyboard aliasing for the Cancel button.
    SetDialogTracksCursor will enable automatic cursor tracking, causing the cursor to
    change to the I-beam shape when it is over the editable text item. }

  osError := SetDialogDefaultItem(modalDlgPtr, iOK);
  osError := SetDialogCancelItem(modalDlgPtr, iCancel);
  osError := SetDialogTracksCursor(modalDlgPtr, true);

  ShowWindow(modalDlgPtr);

{ Specify new event filter for modal dialog in first parameter of ModalDialog call. }

  repeat

  ModalDialog(ModalFilterUPP(@eventFilterModal), itemHit);
  GetDialogItem(modalDlgPtr, itemHit, itemType, itemHdl, itemRect);
  SetControlValue(ControlHandle(itemHdl), not(GetControlValue(ControlHandle(itemHdl))));

  until ((itemHit = iOK) or (itemHit = iCancel));

  if (itemHit = iOK) then
    begin
    GetDialogItem(modalDlgPtr, iGridSnap, itemType, itemHdl, itemRect);
    gGridSnap := GetControlValue(ControlHandle(itemHdl));

    GetDialogItem(modalDlgPtr, iShowGrid, itemType, itemHdl, itemRect);
    gShowGrid := GetControlValue(ControlHandle(itemHdl));

    GetDialogItem(modalDlgPtr, iShowRulers, itemType, itemHdl, itemRect);
    gShowRule := GetControlValue(ControlHandle(itemHdl));
    end;

  DisposeDialog(modalDlgPtr);

  DoModalDialog := true;
  end;
      {of function DoModalDialog}
```

Step 3 is to add this new application-defined filter function for use by the modal dialog:

```
function EventFilterModal(theDialogPtr : DialogPtr; theEvent : EventRecord;
                                  var itemHit : integer) : boolean;

  var
  handledEvent : integer;
  osError : OSErr;
  standardProc : ModalFilterUPP;
  oldPort : GrafPtr;

  begin
  handledEvent := false;

  if ((theEvent.what = updateEvt) and (WindowPtr(theEvent .message) <> theDialogPtr)) then
    begin
    DoUpdate(theEvent);
```

```
    end
  else begin
    GetPort(oldPort);
    SetPort(theDialogPtr);

    { In order for the SetDialogDefaultItem, SetDialogCancelItem, and
      SetDialogTracksCursor calls to work, you must call the standard filter procedure.}

    osError := GetStdFilterProc(standardProc);
    if not (osError) then
      handledEvent = ModalFilterUPP(standardProc) (theDialogPtr, theEvent, itemHit);

    SetPort(oldPort);
    end;

  if (handledEvent <> 0) then
    EventFilterModal := true
  else EventFilterModal := false;

  end;
      {of function EventFilterModal}
```

Creating 'ALRT', 'DLOG', and 'DITL' Resources Using ResEdit

When learning to create the major resource types in ResEdit, it is recommended that you open Macintosh C to the page containing the relevant example resource definition in Rez input format and relate what you are doing within ResEdit to that definition. Accordingly, the methodology used in the following is to "walk through" selected 'ALRT', 'DLOG', and 'DITL' resources for the DialogsAndAlerts demonstration program, relating what you see in ResEdit to the example definitions in this chapter.

Open the chap06pascal_demo demonstration program folder and double-click on the DialogsAndAlerts.µ.rsrc icon to start ResEdit and open DialogsAndAlerts.µ.rsrc.

The **DialogsAndAlerts.µ.rsrc** window opens.

'ALRT' Resource

Double-click the ALRT icon. The **ALRTs from DialogsAndAlerts.µ.rsrc** window opens. Double-click the list entry for ID = 128. The **ALRT ID = 128 from DialogsAndAlerts.µ.rsrc** window opens.

The following relates the example 'ALRT' resource in Rez input format in this chapter to the ResEdit display and interface:

`resource 'ALRT'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **ALRT** was clicked, and the dialog's **OK** button was clicked.
`(kSaveAlertID,`	`kSaveAlertID` is the 'ALRT' resource ID (128). Choose **Resource/Get Resource Info**. The **Info for ALRT 128 ...** window opens. Note the editable text item titled **ID:**. This is where you set the 'ALRT' resource ID. (ResEdit automatically assigns 128 as the 'ALRT' resource ID of the first 'ALRT' resource you create.)
`purgeable)`	While the **Info for ALRT 128 ...** window is open, compare the **Attributes:** check boxes to the Resource Attributes table at Chapter 1. Note that the **Purgeable** checkbox is checked. Close the **Info for ALRT 128 ...** window.
`{94,80,183,438}`	In the **ALRT ID = 128 ...** window, note the **Top**, **Left**, **Bottom**, and **Right** items at the bottom left. (Also note that, in the **ALRT** menu, you can change the last two items to display **Height** and **Width** if you so desire.)
`kAlertItemList`	The resource ID for the item list ('DITL') resource (128). Note the **DITL ID:** item at the right of the window.

`OK,visible,sound1,` `OK,visible,sound1,` `OK,visible,sound1,` `OK,visible,sound1,`	4th, 3rd, 2nd, and 1st alert stages. Choose **ALRT/Set'ALRT' Stage Info...** and note, in turn: • the **Default button/OK/Cancel** checkboxes, • the **Alert box/Visible** checkboxes, and • the **Sounds** clickable items, against the four stages.
`alertPosition...`	Choose **ALRT/Auto Position...** and note the items chosen in the two pop-up menus.

You might also further explore the ResEdit display options by choosing **ALRT/Preview at Full Size**, and the various items in the **MiniScreen** menu.

Note that, when you click on the **Color: Custom** radio button at the right of the **ALRT ID = 128 ...** window, five items appear which enable you to specify colours for the various elements of the alert window. If you were to save the resource with this radio button set, ResEdit would automatically create a `'actb'` (alert color table) resource with the same resource ID as the associated `'ALRT'` resource.

Close the **ALRT ID = 128 ...** window. Close the **ALRTs from DialogsAndAlerts.µ.rsrc** window.

`'DLOG'` Resources

Double-click the DLOG icon. The **DLOGs from DialogsAndAlerts.µ.rsrc** window opens. Several `'DLOG'` resources (IDs 129 to 131) appear in the list. These are, in sequence, the `'DLOG'` resources for:

- The modal dialog (ID 129).
- The movable modal dialog (ID 130).
- The modeless dialog (IDs 131).

Double-click the entry for the modal dialog (ID 129). The **DLOG ID = 129 from DialogsAndAlerts.µ.rsrc** window opens.

The following relates the example `'DLOG'` resource in Rez input format in this chapter to the ResEdit display and interface:

`resource 'DLOG'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **DLOG** was clicked, and the dialog's **OK** button was clicked.
`(kSpellCheckID,`	`kSpellCheckID` is the `'DLOG'` resource ID (129). Choose **Resource/Get Resource Info**. The **Info for DLOG 129 ...** window opens. Note the editable text item titled **ID:**. This is where you set the `'DLOG'` resource ID. (ResEdit automatically assigns 128 as the `'DLOG'` resource ID of the first `'DLOG'` resource you create.)
`purgeable)`	While the **Info for DLOG 129 ...** window is open, compare the **Attributes:** check boxes to the Resource Attributes table at Chapter 1. Note that the **Purgeable** checkbox is checked. Close the **Info for DLOG 129 ...** window.
`{62,184,216,448},`	In the **DLOG ID = 129 ...** window, note the **Top**, **Left**, **Bottom**, and **Right** items at the bottom left. (Note also that, in the **DLOG** menu, you can change the last two items to display **Height** and **Width** if you so desire.)
`dBoxProc,`	Note that, in the row of window icons at the top of the window, the `dBoxProc` (1) window type is highlighted. Note also that, when you choose **DLOG/Set 'DLOG' Characteristics...**, the **ProcID:** item in the opened dialog box shows 1. (You can set the desired Window Definition ID either here or by clicking the appropriate icon at the top of the window.) Close the dialog.

`invisible,`	Back in the **DLOG ID = 128 ...** window, note the check box titled **Initially Visible** at the right.
`noGoAway,`	Note the check box titled **Close Box** at the right.
`kSpellCheckDITL...,`	Note the editable text item **DITL ID:** at the right of the window. This is where you enter the ID of the `'DITL'` resource to be associated with this dialog.
`"SpellCheck Op...",`	Choose **DLOG/Set 'DLOG' Characteristics...**. Note the editable text item **Window title:**. Close the dialog.
`staggerParent...`	Choose **DLOG/Auto Position...** and note the items chosen in the two pop-up menus.

You might also further explore the ResEdit display options by choosing **DLOG/Preview at Full Size**, and the various items in the **MiniScreen** menu.

Note that, when you click on the **Color: Custom** radio button at the right of the **DLOG ID = 129 ...** window, five items appear which enable you to specify colours for the various elements of the window. If you were to save the resource with this radio button set, ResEdit would automatically create a `'dctb'` (dialog color table) resource with the same resource ID as the associated `'DLOG'` resource.

Close the **DLOG ID = 128 ...** window. Close the **DLOGs from DialogsAndAlerts.µ.rsrc** window.

`'DITL'` Resources

Double-click the DITL icon. The **DITLs from DialogsAndAlerts.µ.rsrc** window opens. Several `'DITL'` resources (IDs 128 to 131) appear in the list. These are, in sequence, the `'DITL'` resources for:

- The alert (ID 128).
- The modal dialog (ID 129).
- The movable modal dialog (ID 130).
- The modeless dialog (IDs 131).

Double-click the entry for the modeless dialog (ID 131). The **DITL ID = 131 from DialogsAndAlerts.µ.rsrc** window opens.

The following relates the example `'DITL'` resource in Rez input format in this chapter to the ResEdit display and interface:

`resource 'DITL'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **DITL** was clicked, and the dialog's **OK** button was clicked.
`(kAboutBoxDITL,`	`kAboutBoxDITL` is the `'DITL'` resource ID (131). Choose **Resource/Get Resource Info**. The **Info for DITL 131 ...** window opens. Note the editable text item titled **ID:**. This is where you set the `'DITL'` resource ID. (ResEdit automatically assigns 128 as the `'DITL'` resource ID of the first `'DITL'` resource you create.)
`purgeable)`	While the **Info for DITL 131 ...** window is open, compare the **Attributes:** check boxes to the Resource Attributes table at Chapter 1. Note that the **Purgeable** checkbox is checked. Close the **Info for DITL 131 ...** window.
`{86,201,106,259},`	The display rectangle. In the **DITL ID = 131 from DialogsAndAlerts.µ.rsrc** window, drag item #3 out of the way to fully reveal item #1. (Item #1 was created by dragging a button icon from the item palette roughly into position in the window.) Double click on item #1. The **Edit DITL item #1 ...** window opens. Note the **Top**, **Left**, **Bottom**, and **Right** items. (Also note that, in the **Item** menu, you can change the latter two items to display **Height** and **Width** if you so desire.)

`Button {`	This was established by the icon dragged into the **DITL ID = 131 from DialogsAndAlerts.µ.rsrc** window from the item palette. (However, note that, in the popup menu at the left, the item type can be changed.)
`enabled.`	Note the **Enabled** checkbox at lower left.
`"OK" },`	Note the editable text item **Text**. Close the **Edit DITL item #1 ...** window.
`{10,20,42,52},`	In this case, the Icon item from the item palette was dragged roughly into position in the window . Double-click item #2 to open the **Edit DITL item #2 ...** window. Note the **Top**, **Left**, **Height**, and **Width** values.
`Icon {`	This was established by the Icon icon dragged into the **DITL ID = 131 from DialogsAndAlerts.µ.rsrc** window from the item palette.
`disabled,`	Note the **Enabled** checkbox at lower left.
`kAboutIconID },`	Note the **Resource ID** item. Close the **Edit DITL item #1 ...** window. Close the **DITL ID = 131 from DialogsAndAlerts.µ.rsrc** window.

Close the DITLs from **DialogsAndAlerts.µ.rsrc** window. Close the **DialogsAndAlerts.µ.rsrc** window without saving.