

8

DIALOGS AND ALERTS

Includes Demonstration Program DialogsAndAlerts

Introduction

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

Types of Alerts and Alert Boxes

When an alert condition occurs, and depending on the nature of that condition, your application can simply play an alert sound¹ 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.

Types of Alert Box

There are two types of alert box: the **modal alert box** and the **movable modal alert box**. Both types are shown at Fig 1.

Modal Alert Box

The fixed-position modal alert box places the user in the state, or **mode**, of being able to work only inside the alert box. The only response the user receives when clicking anywhere outside the alert box is the alert sound, and the user is not be able to bring another application to the front before first dismissing the alert box.

Movable Modal Alert Box

Movable modal alert boxes retain the essentially modal characteristic of their fixed-position counterpart, the main differences being that they allow the user to:

- Drag the alert box so as to uncover obscured areas of an underlying window.

¹ 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 function 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.

- 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.

Historical Note

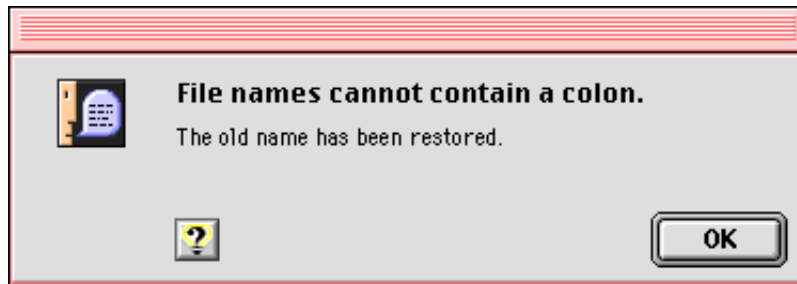
The movable modal alert box was introduced with OS 8 and the Appearance Manager.

Generally speaking, the movable modal alert box is more appropriate for use in a multi-tasking environment. The modal alert box should be used only where there is justification for denying the user access to other running applications before it is dismissed.

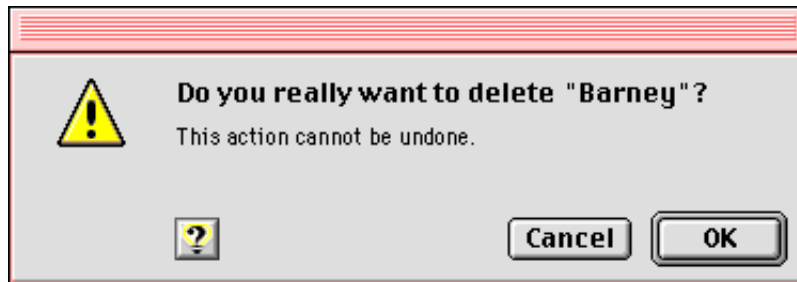
Levels of Alerts

An alert box can display either one of three levels of alert, depending on the nature of the situation the alert box is reporting to the user. The three levels of alert, which are identified by icons supplied automatically by the system, are as follows:

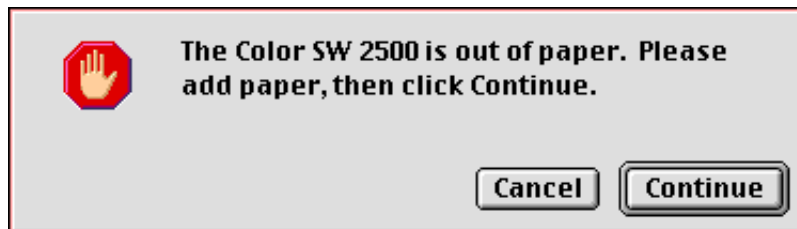
- **Note Alert.** The note alert is used to inform users of an occurrence which will not have serious consequences. Usually, a note alert simply offers information. Sometimes, a note alert may ask a simple question and provide, via the push buttons, a choice of responses.



A NOTE ALERT DISPLAYED IN A MOVABLE MODAL ALERT BOX



A CAUTION ALERT DISPLAYED IN A MOVABLE MODAL ALERT BOX



A STOP ALERT DISPLAYED IN A MODAL ALERT BOX

FIG 1 - MODAL AND MOVABLE MODAL ALERT BOXES

- **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 push buttons, with a choice of whether to continue or stop the action.

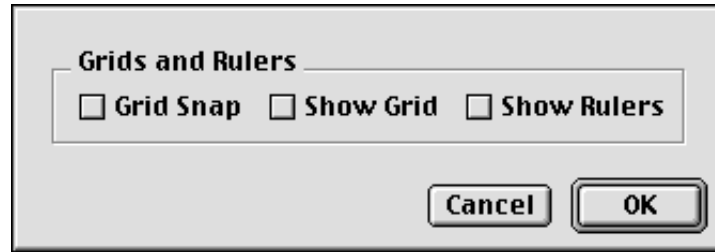
- *Stop Alert.* The stop alert is used to inform the user that a problem or situation is so serious that the action cannot be completed.

Custom Alert Boxes

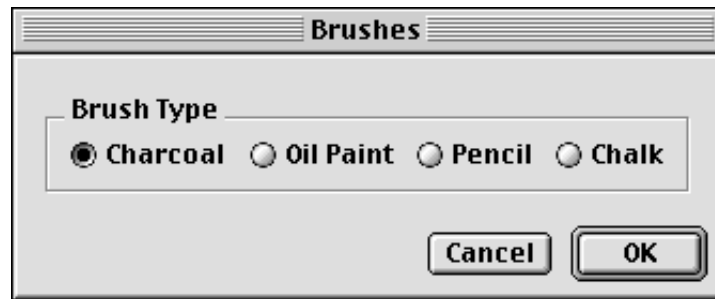
You can also create **custom alert boxes**, which might contain your own icons (or, possibly, no icons). Custom alert boxes are typically used for **About...** boxes.

Types Of Dialogs Boxes

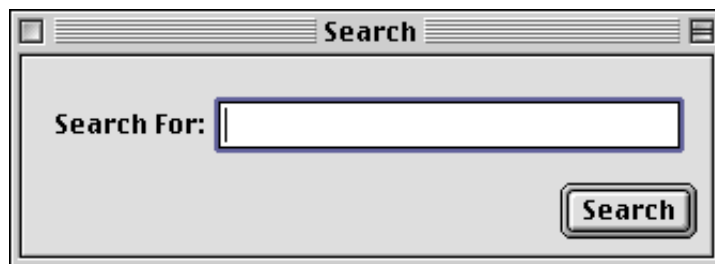
There are three types of dialog box, all of which are illustrated in the examples at Fig 2.



MODAL DIALOG BOX



MOVABLE MODAL DIALOG BOX



MODELESS DIALOG BOX

FIG 2 - MODAL, MOVABLE MODAL, AND MODELESS 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 a modal alert box except that it may contain other types of controls in addition to push buttons.

Modal dialog boxes really belong to the era before the Mac OS became a multi-tasking operating system. Accordingly, they should be used only where there is justification for denying the user access to other running applications before they are dismissed.

Movable Modal Dialog Boxes

Movable modal dialog boxes retain the essentially modal characteristic of their fixed-position counterpart, the main differences being that they allow the user to:

- Drag the dialog box so as to uncover obscured areas of an underlying window.
- 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 of a movable modal dialog box visually suggests to the user that the dialog box is modal rather than modeless.

Modeless Dialog Boxes

Modeless dialog boxes look and behave very like document windows, except for their interior colour/pattern and the one-pixel frame just inside the window frame. Modeless dialog boxes 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 modeless dialog 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.

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.

Window Types For Alerts and Dialogs

Fig 3 shows the seven available Appearance-compliant window types for alerts and dialogs and the constants that represent the window definition IDs for those types. Note that modeless dialogs are a special case in that a normal document window type is used.

The window definition ID is derived by multiplying the resource ID of the WDEF by 16 and adding the variation code to the result, as is shown in the following:

<i>WDEF Resource ID</i>	<i>Variation Code</i>	<i>Window Definition ID (Value)</i>	<i>Window Definition ID (Constant)</i>
65	0	$65 * 16 + 0 = 1040$	kWindowPlainDialogProc
65	1	$65 * 16 + 1 = 1041$	kWindowShadowDialogProc
65	2	$65 * 16 + 2 = 1042$	kWindowModalDialogProc
65	3	$65 * 16 + 3 = 1043$	kWindowMovableModalDialogProc
65	4	$65 * 16 + 4 = 1044$	kWindowAlertProc
65	5	$65 * 16 + 5 = 1045$	kWindowMovableAlertProc
65	6	$65 * 16 + 6 = 1046$	kWindowMovableModalGrowProc
64	0	$64 * 16 + 0 = 1024$	kWindowDocumentProc



FIG 3 - WINDOW TYPES FOR DIALOGS AND ALERTS

Historical Note

The old pre-Mac OS 8, pre-Appearance Manager window types, and their Appearance-compliant equivalents, are as follows.

<i>Pre-Appearance</i>	<i>Appearance-Compliant</i>	<i>Description</i>
(None)	kWindowAlertProc	Modal alert box.
dBoxProc	kWindowModalDialogProc	Modal dialog box.
plainDBox	kWindowPlainDialogProc	Modal dialog box.
altDBoxProc	kWindowShadowDialogProc	Modal dialog box, shadow.
(None)	kWindowMovableAlertProc	Movable modal alert box.
movableDBoxProc	kWindowMovableModalDialogProc	Movable modal dialog box.
noGrowDocProc	kWindowDocumentProc	Document window and modeless dialog box.

Items in Alert and Dialog Boxes

You use **item lists**, which are resources of type 'DITL', to specify the **items** to appear in alert boxes and dialog boxes.

Alert boxes should usually contain only informative text and push button controls. Dialog boxes may contain informative or instructional text and controls.

Default Push Buttons

Your application should specify a **default push button** for every alert and dialog box. The default push button, which is visually identified by the default ring drawn around it, should be the one the user is more likely to click in most circumstances. However, if the most likely choice is at all destructive (for example, erasing a disk or deleting a file), you should consider defining the **Cancel** button as the default.

Removal of Alert and Dialog Boxes

The Dialog Manager automatically removes and disposes of an alert box when the user clicks its **OK** or **Cancel** push button.

Your application should remove and dispose of a modal dialog box or movable modal dialog box only when the user clicks its **OK** or **Cancel** push button.

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. (Typically, a modeless dialog is simply hidden, not disposed of, when the user clicks the close box or chooses **Close**.)

Creating Alert Boxes

Alert boxes may be created in one of two ways:

- Using the functions `Alert`, `NoteAlert`, `CautionAlert` and `StopAlert`, which take descriptive information about the alert from **alert** ('ALRT') and **extended alert** ('alrx') **resources**. The resource ID of the 'ALRT' and 'alrx' resources must be the same, and is passed in the first parameter of these functions. You specify whether the alert is to be displayed in a modal alert box or a movable modal alert box by setting a particular flag in the 'alrx' resource.
- Using the function `StandardAlert`, which optionally takes additional descriptive information about the alert from a **standard alert structure** (a structure of type `AlertStdAlertParamRec`). You specify whether the alert is to be a plain, note, caution, or stop alert in the first parameter of the `StandardAlert` call. You specify whether the alert is to be displayed in a modal alert box or a movable modal alert box by assigning a value to the first field of the standard alert structure.

Historical Note

The extended alert resource, the `StandardAlert` function, and the standard alert structure were introduced with Mac OS 8 and the Appearance Manager.

Regardless of which method is used to create the alert box, the standard note, caution, and stop icons automatically appear in the upper-left corner of alert boxes displaying, respectively, note, caution and stop alerts. When the `Alert` function is used, or when a plain alert is created using `StandardAlert`, no icon is displayed automatically, though you can display your own icon.

'ALRT' and 'alrx' Resources

When creating resources with Resorcerer, it is advisable that you refer to a diagram and description of the structure of the resource and relate that to the various items in the Resorcerer editing windows. Accordingly, the following describes the structure of the resources associated with the creation of alert boxes.

Structure of a Compiled 'ALRT' Resource

Fig 4 shows the structure of a compiled 'ALRT' resource.

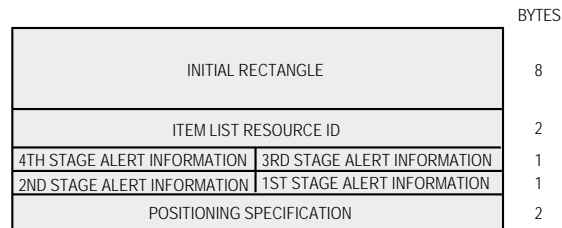


FIG 4 - STRUCTURE OF A COMPILED ALERT ('ALRT') RESOURCE

The following describes the fields of the 'ALRT' resource:

<i>Field</i>	<i>Description</i>
INITIAL RECTANGLE	A rectangle which determines the dimensions of the content region of the alert box's window and, possibly, the position of the rectangle. (The last field of the resource usually specifies a position for the alert.)
ITEM LIST RESOURCE ID	The ID of the item list resource that specifies the items, such as press buttons and static text, that are to be displayed in the alert box.
4TH STAGE ALERT INFORMATION	Specifies the response when the user repeats the action that invokes the alert four or more consecutive times. (See Alert Stages, below.)
3RD STAGE ALERT INFORMATION	Specifies the response when the user repeats the action that invokes the alert three consecutive times. (See Alert Stages, below.)
2ND STAGE ALERT INFORMATION	Specifies the response when the user repeats the action that invokes the alert two consecutive times. (See Alert Stages, below.)
1ST STAGE ALERT INFORMATION	Specifies the response for the first time the user invokes the action that invokes the alert. (See Alert Stages, below.)
POSITIONING SPECIFICATION	Specifies the position of the alert box on the screen. 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. (See Positioning Specification, below.)

Alert Stages

You can 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, you might specify that the system alert sound be played the first time the user makes the mistake and, subsequently, that the alert box be displayed as well. Note that every occurrence of the mistake after the fourth is treated as a fourth stage alert.

As will be seen, for each of the four alert stages, you can specify whether the default ring is drawn around the first or second push buttons in the alert, whether the alert box is to be displayed, and whether the system alert sound is to be played one, two, or four times, if at all.

Positioning Specification

The constants for the positioning specification field are as follows:

<i>Constant</i>	<i>Value</i>	<i>Meaning</i>
kWindowDefaultPosition	0x0000	Use initial location.
kWindowAlertPositionMainScreen	0x300A	Alert position on main screen.
kWindowCenterParentWindow	0xA80A	Centre on parent window.
kWindowAlertPositionParentWindow	0xB00A	Alert position on parent window.
kWindowAlertPositionParentWindowScreen	0x700A	Alert position on parent window screen.

Structure of a Compiled 'alrx' Resource

Fig 5 shows the structure of a compiled 'alrx' resource. You use this resource to provide additional features for your alert box, including movable modal behavior, theme-compliant backgrounds and controls, and control embedding hierarchies.

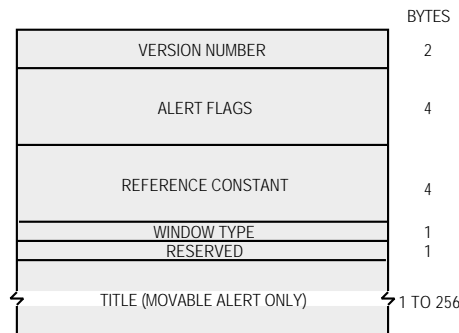


FIG 5 - STRUCTURE OF A COMPILED EXTENDED ALERT ('alrx') RESOURCE

The following describes the main fields of the 'alrx' resource:

<i>Field</i>	<i>Description</i>
ALERT FLAGS	Constants specifying the alert box's Appearance features. (See Alert Feature Flag Constants, below).
REFERENCE CONSTANT	Any value that an application wishes to store here. For example, an application can store a number here that enables it to distinguish between a number of similar alert boxes. Since this information is stored in a window structure within the dialog structure, you can use <code>GetWRefCon</code> to determine this value.
WINDOW TYPE	If this Boolean is set to 1 (true), the Dialog Manager specifies an Appearance-compliant window definition ID directly when drawing the alert box window.
TITLE	A string representing the title of a movable alert box

Alert Feature Flag Constants

You can set the following bits in the alert flags field of an 'alrx' resource to specify the alert box's Appearance-compliant features:

<i>Constant</i>	<i>Bit</i>	<i>Meaning</i>
<code>kAlertFlagsUseThemeBackground</code>	0	If this bit is set, the Dialog Manager sets the alert box's background colour or pattern.
<code>kAlertFlagsUseControlHierarchy</code>	1	If this bit is set, the Dialog Manager creates a root control in the alert box and establishes an embedding hierarchy. All alert items automatically become controls once the embedding hierarchy is established.
<code>kAlertFlagsAlertIsMovable</code>	2	If this bit is set, the alert box is movable modal. The Dialog Manager handles movable modal behavior such as dragging the alert box by its title bar or switching out of the application by clicking in another one.
<code>kAlertFlagsUseThemeControls</code>	3	If this bit is set, the Dialog Manager creates Appearance-compliant controls in the alert box. Otherwise, push buttons, checkboxes, and radio buttons will be displayed in their pre-Appearance forms when system-wide Appearance is selected off.

Creating 'ALRT' and 'alrx' Resources Using Resorcerer

Creating 'ALRT' Resources

Fig 6 shows an 'ALRT' resource being created with Resorcerer.

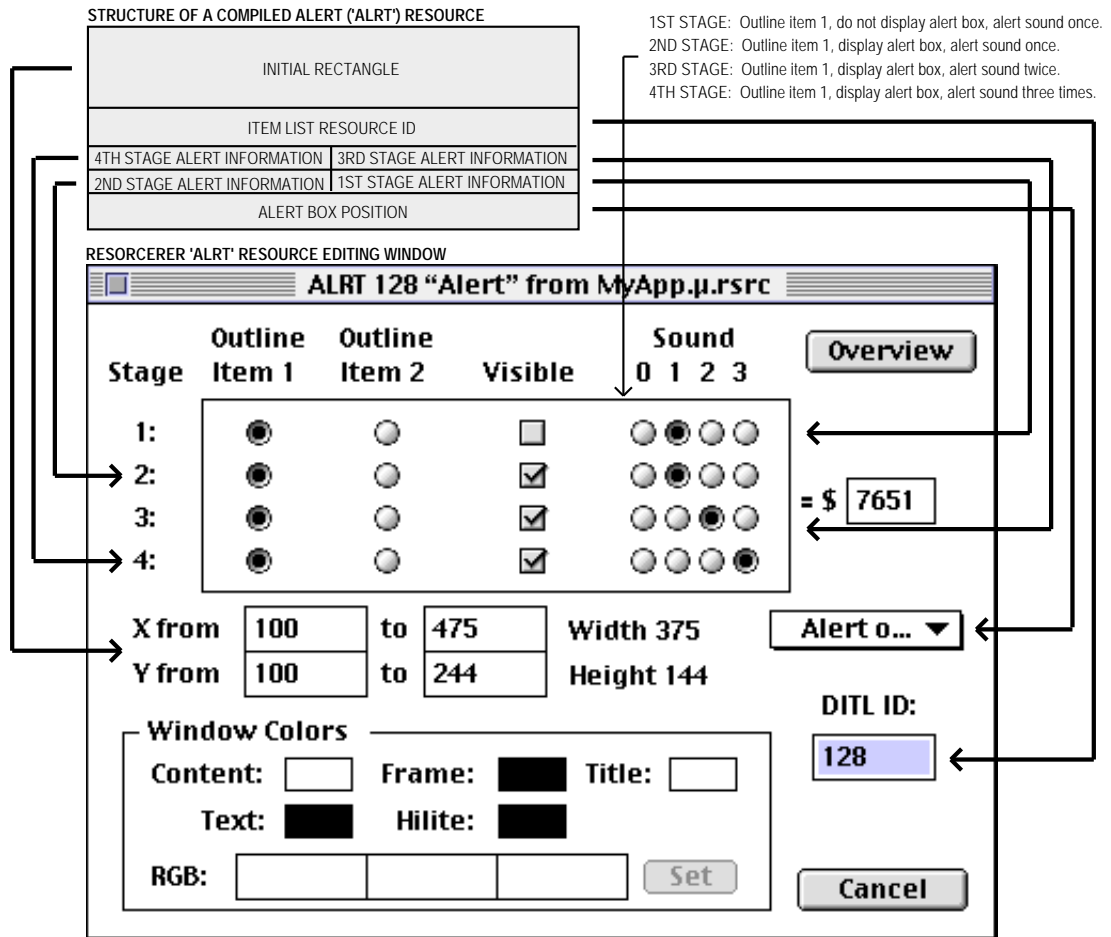


FIG 6 - CREATING AN 'ALRT' RESOURCE USING RESORCERER

Creating 'alrx' Resources

Fig 7 shows an 'alrx' resource being created with Resorcerer. Note that the resource ID of an 'alrx' resource must be the same as the resource ID of the 'ALRT' resource with which it is associated.

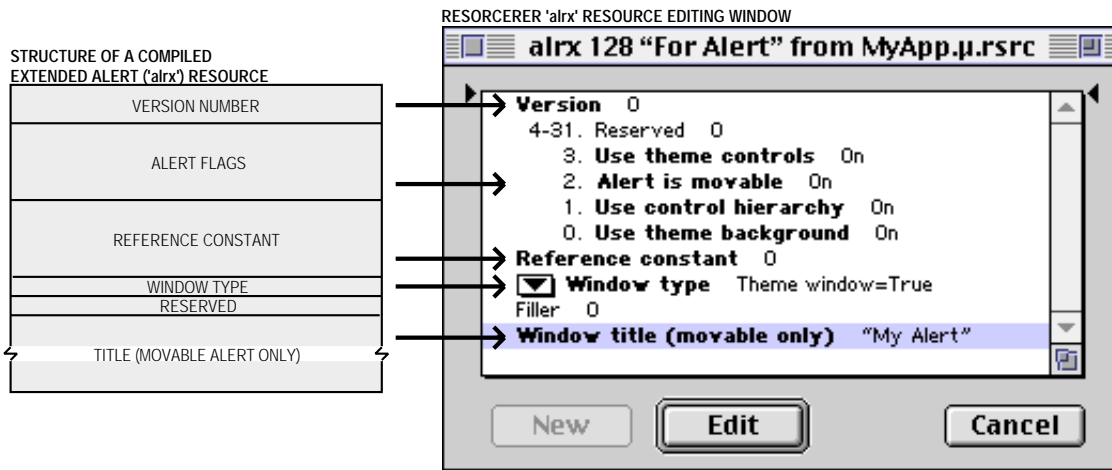


FIG 7 - CREATING AN 'alrx' RESOURCE USING RESORCERER

The StandardAlert Function

When an alert box is created using the function `StandardAlert`, the alert box is automatically sized based on the amount of text passed in to it, and push buttons are automatically sized and located. `StandardAlert` does not provide for alert stages and corresponding alert sounds.

```
OSErr StandardAlert(AlertType inAlertType, StringPtr inError, StringPtr inExplanation,
AlertStdAlertParamPtr inAlertParam, SInt16 *outItemHit);
```

Returns: A result code.

inAlertType A constant indicating the type of alert box you wish to create (modal or movable modal).

inError A pointer to a Pascal string containing the primary error (label) text to display.

inExplanation A pointer to a Pascal string containing the secondary (narrative) text to display. Secondary text is displayed in the small system font. Pass NULL to indicate no secondary text.

inAlertParam A pointer to a standard alert structure (see below). Pass NULL to specify that you do not wish your alert box to incorporate any of the features that the standard alert structure provides.

outItemHit A pointer to an integer that, on output, will contain a value indicating the item number of the push button that was hit.

Standard Alert Structure

The standard alert structure is as follows:

```
struct AlertStdAlertParamRec
{
    Boolean        movable;
    Boolean        helpButton;
    ModalFilterUPP filterProc;
    StringPtr      defaultText;
    StringPtr      cancelText;
    StringPtr      otherText;
    SInt16         defaultButton;
    SInt16         cancelButton;
    UInt16         position;
};
typedef struct AlertStdAlertParamRec AlertStdAlertParamRec;
typedef AlertStdAlertParamRec *AlertStdAlertParamPtr;
```

Field Descriptions

<code>movable</code>	A Boolean value indicating whether or not the alert box is movable.
<code>helpButton</code>	A Boolean value indicating whether or not the alert includes a Help button.
<code>filterProc</code>	You may specify in this parameter a universal procedure pointer to an application-defined event filter function. If you set this parameter to <code>NULL</code> , the Dialog Manager uses the standard event filter function. (See <i>Event Filter Functions For Modal and Movable Modal Alert and Dialog Boxes</i> , below).
<code>defaultText</code>	Text for the push button in the OK push button position. (See <i>Alert Default Text Constants</i> , below). The push button automatically sizes and positions itself in the alert box. To indicate that no push button should be displayed, pass <code>NULL</code> .
<code>cancelText</code>	Text for the push button in the Cancel position. (See <i>Alert Default Text Constants</i> , below.) The push button automatically sizes and positions itself in the alert box. To indicate that no push button should be displayed, pass <code>NULL</code> .
<code>otherText</code>	Text for the push button in leftmost position. (See <i>Alert Default Text Constants</i> , below.) The push button automatically sizes and positions itself in the alert box. To indicate that no push button should be displayed, pass <code>NULL</code> .
<code>defaultButton</code>	Specifies which push button acts as the default push button. (See <i>Alert Button Constants</i> , below.)
<code>cancelButton</code>	Specifies which push button acts as the Cancel push button. Can be <code>0</code> . (See <i>Alert Button Constants</i> , below.)
<code>position.</code>	The alert box position, as defined by a window positioning specification. (See <i>Positioning Specification</i> , above, but note that, in this structure, the constant <code>kWindowDefaultPosition</code> is equivalent to <code>kWindowAlertPositionParentWindowScreen</code> .)

Alert Default Text Constants

You can use these constants in the `defaultText`, `cancelText`, and `otherText` fields of the standard alert structure to specify the default text for the **OK**, **Cancel**, and **Don't Save** push buttons.

<i>Constant</i>	<i>Value</i>	<i>Meaning</i>
<code>kAlertDefaultOKText</code>	-1	The default text for the default (right) push button is "OK" on an English system. The text will vary depending upon the localisation of the user's system. Use this constant in the <code>defaultText</code> field of the standard alert structure.
<code>kAlertDefaultCancelText</code>	-1	The default text for the Cancel (middle) push button is "Cancel" on an English system. The text will vary depending upon the localisation of your system. Use this constant in the <code>cancelText</code> field of the standard alert structure.
<code>kAlertDefaultOtherText</code>	-1	The default text for the third (leftmost) push button is "Don't Save" for an English system. The text will vary depending upon the localisation of the user's system. Use this constant in the <code>otherText</code> field of the standard alert structure.

Alert Push Button Constants

You can use these constants in the `defaultButton` and `cancelButton` fields in the standard alert structure to specify which push buttons act as the default and **Cancel** push buttons in the standard alert structure. These constants are also returned in the `itemHit` parameter of `StandardAlert`.

<i>Constant</i>	<i>Value</i>	<i>Meaning</i>
kAlertStdAlertOKButton	1	The OK push button. The default text for this push button is “OK”.
kAlertStdAlertCancelButton	2	The Cancel push button (optional). The default text for this push button is “Cancel”.
kAlertStdAlertOtherButton	3	A third push button (optional). The default text for this push button is “Don’t Save”.
kAlertStdAlertHelpButton	4	The Help push button (optional).

Creating Dialog Boxes

Dialog boxes may be created in one of two ways:

- Using the function `GetNewDialog`, which takes descriptive information about the dialog from **dialog** ('DLOG') and **extended dialog** ('dlgx') resources. The resource ID of the 'DLOG' and 'dlgx' resources must be the same, and is passed in the first parameter of this function.
- Using `NewDialog`, `NewColorDialog`, or `NewFeaturesDialog`, which take descriptive information passed in the parameters of those functions.

Historical Note

The extended dialog resource and the `NewFeaturesDialog` function were introduced with OS 8 and the Appearance Manager. `NewFeaturesDialog` should be used to create Appearance-compliant dialogs. Unlike `NewDialog` and `NewColorDialog`, it has a flags parameter containing the same flags you would set in an extended dialog resource.

If `NULL` is specified as the second parameter in the `GetNewDialog` call, `GetNewDialog` itself creates a non-relocatable block for the dialog structure. Passing `NULL` is appropriate for modal and movable modal dialog boxes because you should dispose of the dialog box, and hence the associated dialog structure, when the user dismisses the dialog. However, in order to avoid heap fragmentation effects, you should ordinarily allocate your own memory for modeless dialog box dialog structures (just as you would for a window structure) and specify the pointer to that memory block in the second parameter of the `GetNewDialog` call.

Regardless of which method is used to create the dialog, a **dialog structure** and a window structure will be created, and a pointer to the dialog structure will be returned to the calling function.

The Dialog Structure

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

```
struct DialogRecord
{
    WindowRecord window;    // Dialog's window record.
    Handle items;          // Item list resource.
    TEHandle textH;        // Current editable text item.
    SInt16 editField;      // Editable text item number minus 1.
    SInt16 editOpen;       // (Used internally.)
    SInt16 aDefItem;       // Default push button item number.
};
typedef struct DialogRecord DialogRecord;
typedef DialogRecord *DialogPeek;
```

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

'DLOG' and 'dlgx' Resources

Structure of a Compiled 'DLOG' Resource

Fig 8 shows the structure of a compiled 'DLOG' resource and how it "feeds" the dialog structure.

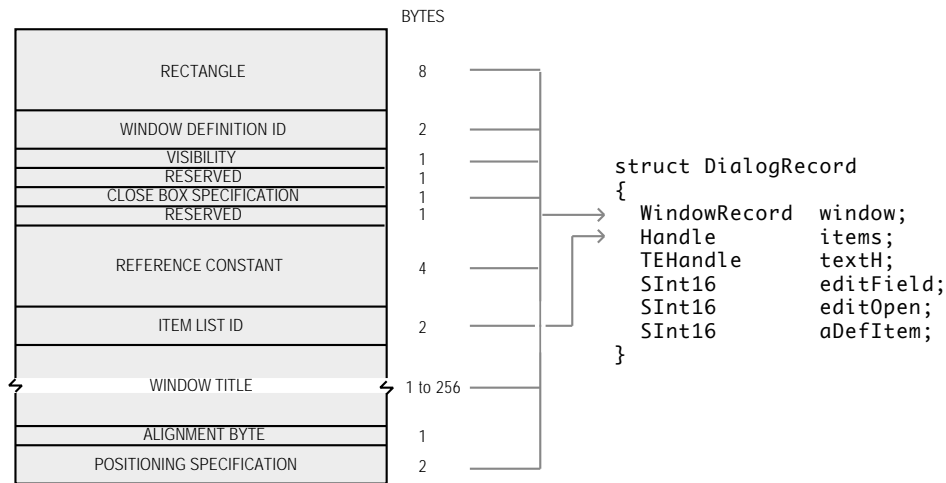


FIG 8 - STRUCTURE OF A COMPILED DIALOG ('DLOG') RESOURCE

The following describes the main fields of the 'DLOG' resource:

<i>Field</i>	<i>Description</i>
RECTANGLE	Determines the dialog box's dimensions and, possibly, its position. (The last field of the resource usually specifies a position for the dialog box.)
WINDOW DEFINITION ID	The window definition ID. (See Window Types For Alerts and Dialogs, above.)
VISIBILITY	If set to the value of 1, the Dialog Manager displays this dialog box as soon as you call the <code>GetNewDialog</code> function. If set to the value of 0, the Dialog Manager does not display the dialog box until you call <code>ShowWindow</code> .
CLOSE BOX SPECIFICATION	This specifies whether to draw a close box. Normally, this is set to the value of 1 only for a modeless dialog box.
REFERENCE CONSTANT	Contains any value an application stores here. For example, an application can store a number that represents a dialog box type, or it can store a handle to a record that maintains state information about the dialog box or other window types. An application can use the <code>SetWRefCon</code> function to change this value and <code>GetWRefCon</code> to determine the current value.
ITEM LIST ID	The resource ID of the item list resource associated with this dialog box.
WINDOW TITLE	A Pascal string displayed in the dialog box's title bar (modeless and movable modal dialog boxes only).
POSITIONING SPECIFICATION	Specifies the position of the dialog box on the screen. If a positioning constant is not provided, the Dialog Manager places the dialog box at the global coordinates you specify for the dialog box's rectangle. The same positioning constants as apply in the case of an alert box apply. (See Positioning Specification, above.)

Structure of a Compiled 'dlgx' Resource

Fig 9 shows the structure of a compiled 'dlgx' resource. This resource allows you to provide additional features for your dialog box, including movable modal behavior, theme-compliant backgrounds and controls, and embedding hierarchies.

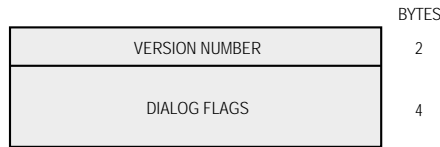


FIG 9 - STRUCTURE OF A COMPILED DIALOG ('dlgx') RESOURCE

The following describes the main field of the 'dlgx' resource:

<i>Field</i>	<i>Description</i>
DIALOG FLAGS	Constants that specify the dialog box's Appearance features. (See Dialog Feature Flag Constants, below.)

Dialog Feature Flag Constants

You can set the following bits in the dialog flags field of a 'dlgx' resource to specify the dialog box's Appearance-compliant features:

<i>Constant</i>	<i>Bit</i>	<i>Meaning</i>
kDialogFlagsUseThemeBackground	0	If this bit is set, the Dialog Manager sets the dialog box's background colour or pattern.
kDialogFlagsUseControlHierarchy	1	If this bit is set, the Dialog Manager creates a root control in the dialog box and establishes an embedding hierarchy. Any dialog items automatically become controls once the embedding hierarchy is established.
kDialogFlagsHandleMovableModal	2	If this bit is set, the dialog box is movable modal (in which case you must use kWindowMovableModalDialogProc window definition ID). The Dialog Manager handles movable modal behavior such as dragging the dialog box by its title bar or switching out of the application by clicking in another one.
kDialogFlagsUseThemeControls	3	If this bit is set, the Dialog Manager creates Appearance-compliant controls in the dialog box directly. Otherwise, push buttons, checkboxes, and radio buttons will be displayed in their pre-Appearance forms when system-wide Appearance is selected off.

Creating 'DLOG' and 'dlgx' Resources Using Resorcerer

Creating 'dlgx' Resources

Fig 10 shows a 'dlgx' resource being created with Resorcerer.

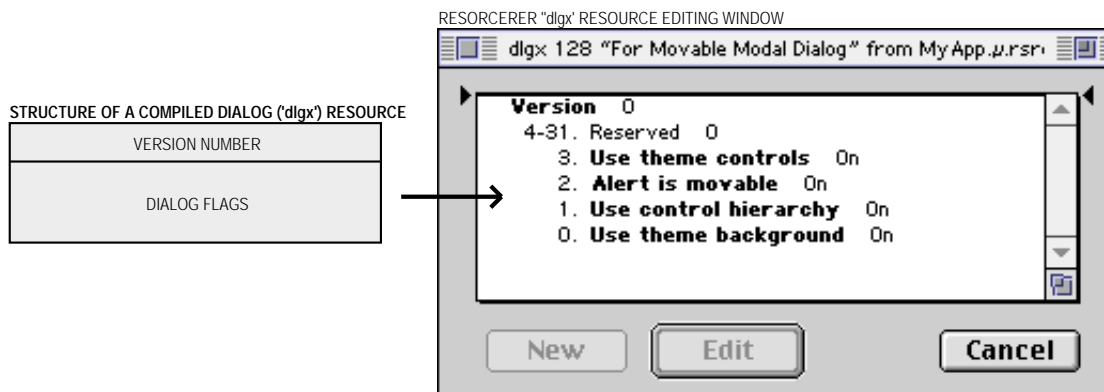


FIG 10 - CREATING A 'dlgx' RESOURCE USING RESORCERER

Creating 'DLOG' Resources

Fig 11 shows a 'DLOG' resource being created with Resorcerer.

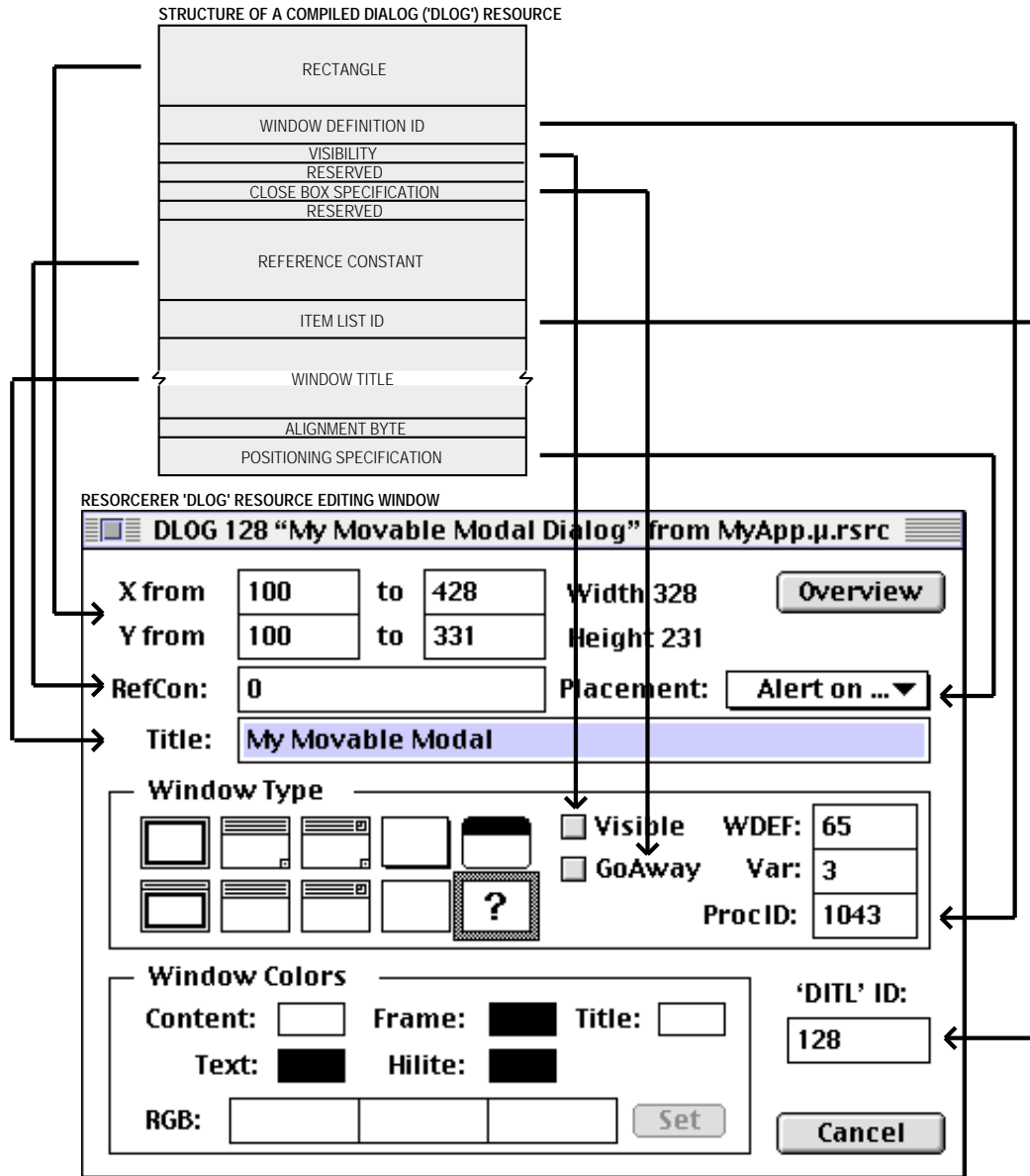


FIG 11 - CREATING A 'DLOG' RESOURCE USING RESORCERER

The NewFeaturesDialog Function

As previously stated, the function `NewFeaturesDialog` creates a dialog from the information passed in its parameters.

```
DialogPtr NewFeaturesDialog(void *inStorage, const Rect *inBoundsRect,
    ConstStr255Param inTitle, Boolean inIsVisible, SInt16 inProcID,
    WindowPtr inBehind, Boolean inGoAwayFlag, SInt32 inRefCon,
    Handle inItemListHandle, UInt32 inFlags);
```

Returns: A pointer to the new dialog box, or NULL if the dialog box is not created

<i>inStorage</i>	A pointer to the memory for the dialog structure. If you set this parameter to <code>NULL</code> , the Dialog Manager automatically allocates a nonrelocatable block in your application heap.
<i>inBoundsRect</i>	A pointer to a rectangle, given in global coordinates, that determines the size and position of the dialog box. These coordinates specify the upper-left and lower-right corners of the dialog box.
<i>inTitle</i>	A pointer to a text string used for the title of a modeless or movable modal dialog box. You can specify an empty string ² (not <code>NULL</code>) for a title bar that contains no text.
<i>isVisible</i>	A flag that specifies whether the dialog box should be drawn on the screen immediately. If you set this parameter to <code>false</code> , the dialog box is not drawn until your application calls <code>ShowWindow</code> to display it.
<i>inProcID</i>	The window definition ID for the type of dialog box. Use the <code>kWindowModalDialogProc</code> constant to specify modal dialog boxes, the <code>kWindowMovableModalDialogProc</code> constant to specify movable modal dialog boxes, and the <code>kWindowDocumentProc</code> constant to specify modeless dialog boxes.
<i>inBehind</i>	A pointer to the window behind which the dialog box is to be placed on the desktop. Set this parameter to the window pointer (<code>WindowPtr</code>)-1 to bring the dialog box in front of all other windows.
<i>inGoAwayFlag</i>	A Boolean value. If <code>true</code> , specifies that an active modeless dialog box has a close box in its title bar.
<i>inRefCon</i>	A value that the Dialog Manager uses to set the <code>refCon</code> field of the dialog box's window structure. Your application may store any value here for any purpose. For example, your application can store a number that represents a dialog box type, or it can store a handle to a structure that maintains state information about the dialog box. You can use the Window Manager function <code>SetWRefCon</code> to change this value and <code>GetWRefCon</code> to determine its current value.
<i>inItemListItemHandle</i>	A handle to an item list resource for the dialog box. You can get the handle by calling <code>GetResource</code> to read the item list resource into memory.
<i>inFlags</i>	An unsigned 32-bit mask specifying the dialog box's Appearance-compliant feature flags. (See Dialog Feature Flag Constants, above.)

Although the `inItemListItemHandle` parameter specifies an item list ('DITL') resource for the dialog box, the corresponding dialog font table ('dftb') resource (see below) is not automatically accessed. You must explicitly set the dialog box's control font styles individually.

Items for Alert and Dialog Boxes

Preamble - Dialog Manager Primitives

Alert and dialog boxes contain **items**, such as push buttons, radio buttons, and checkboxes. Prior to the introduction of Mac OS 8 and the Appearance Manager, an actual control could be an item; however, items such as push buttons and radio buttons were not controls as such but rather **Dialog Manager primitives**.

² In C, you specify an empty string by two double quotation marks ("").

These primitives may still be specified in **item list resources** (see below). However, under Appearance, when a root control has been created for the alert or dialog window, thus creating an embedding hierarchy for controls, the Dialog Manager replaces any primitives in the alert or dialog box with their control counterparts (except for the primitive called a **user item**).

In the era of the Appearance Manager, the situation where all items in an alert or dialog are controls has many advantages. For example, all controls within the alert or dialog can be activated and deactivated by simply activating and deactivating the root control. This includes edit text fields and static text fields. (It was previously not possible to display the old edit text and static text primitives in the dimmed deactivated mode.)

The continued use of the Dialog Manager primitives is not inconsistent with the concept of Appearance-compliant alert and dialog boxes provided that an embedding hierarchy has been established, thus causing the Dialog Manager to replace the primitives with (Appearance-compliant) controls.

The primitives, and their control equivalents, are as follows:

<i>Dialog Manager Primitive</i>	<i>Control Equivalent</i>	<i>Comment</i>
Button.	Press button.	
Radio Button.	Radio button.	Used in dialog boxes only.
Checkbox.	Checkbox.	Used in dialog boxes only.
Edit Text.	Edit text field.	Used in dialog boxes only.
Static Text.	Static text field.	
Icon. (An icon whose black and white resource was stored in an 'ICON' resource and whose colour version is stored in a 'icn' resource with the same ID.)	Icon control (no track variant).	
Picture. (Picture stored in a 'PICT' resource.)	Picture control (no track variant).	
User Item. (An application-defined item. For example, an application-defined drawing function could be installed in a user item.)		Used in dialog boxes only.

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. 'DITL' resources should be marked as purgeable.

Within a 'DITL' resource for an alert box you can specify static text, button, icon and picture primitives, or their control equivalents. In dialog boxes, any of the primitives, and any control, may be specified.

Items are usually referred to by their position in the item list, that is, by their **item number**.

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 resources for use in individual dialog boxes.

Fig 12 shows the structure of a compiled 'DITL' resource and one of its constituent items, in this case a control item.

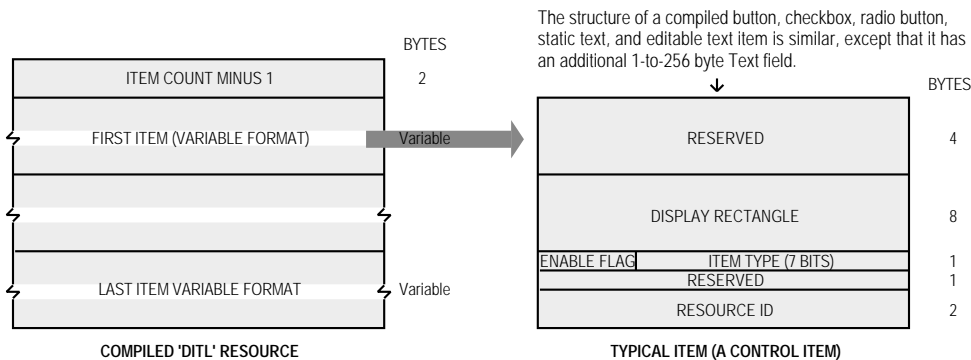


FIG 12 - STRUCTURE OF A COMPILED ITEM LIST ('DITL') RESOURCE AND A TYPICAL ITEM

The following describes the fields of the 'DITL' resource and the control item:

<i>Field</i>	<i>Description</i>
ITEM COUNT MINUS 1	This value is one less than the number of items defined in the resource.
FIRST ITEM ... LAST ITEM	The format of each item depends on its type.
DISPLAY RECTANGLE	Determines the size and location of the item in the alert box or dialog box. The rectangle is specified in coordinates local to the alert box or dialog box. (See Display Rectangles, below.)
ENABLE FLAG	Specifies whether the item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever a mouse-down event occurs inside the item.
ITEM TYPE	Specifies the item type.
RESOURCE ID	For a control item, this is the resource ID of the 'CNTL' resource.

Display Rectangles

For controls, the item's display rectangle becomes the control's enclosing rectangle. To match a control's enclosing rectangle to its own display rectangle, specify an enclosing rectangle in the 'CNTL' resource identical to the display rectangle specified in the 'DITL' resource.³⁴ Other important aspects of display rectangles are as follows:

- **Edit Text Fields.** For an edit text field, the display rectangle becomes the TextEdit **destination rectangle** and **view rectangle** (see Chapter 19 — 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.
- **Static Text Fields.** For a static text field, the Dialog manager draws the text within the display rectangle just as it draws edit text fields.
- **Icon and Picture Controls.** For an icon or 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.

Creating a 'DITL' Resource Using Resorcerer

Fig 13 shows a 'DITL' resource being created with Resorcerer. Two items are being edited (Item 1 and Item 2). Item 1 is a Dialog Manager primitive. Item 2 is a control.

³ When an item is a control defined in a control resource, the rectangle added to the update region is the display rectangle defined in the 'CNTL' resource, not the display rectangle specified in the 'DITL' resource.

⁴ Resorcerer has a Preferences setting which forces conformity between the display rectangle specified in the 'DITL' resource and the display rectangle specified in the 'CNTL' resource.

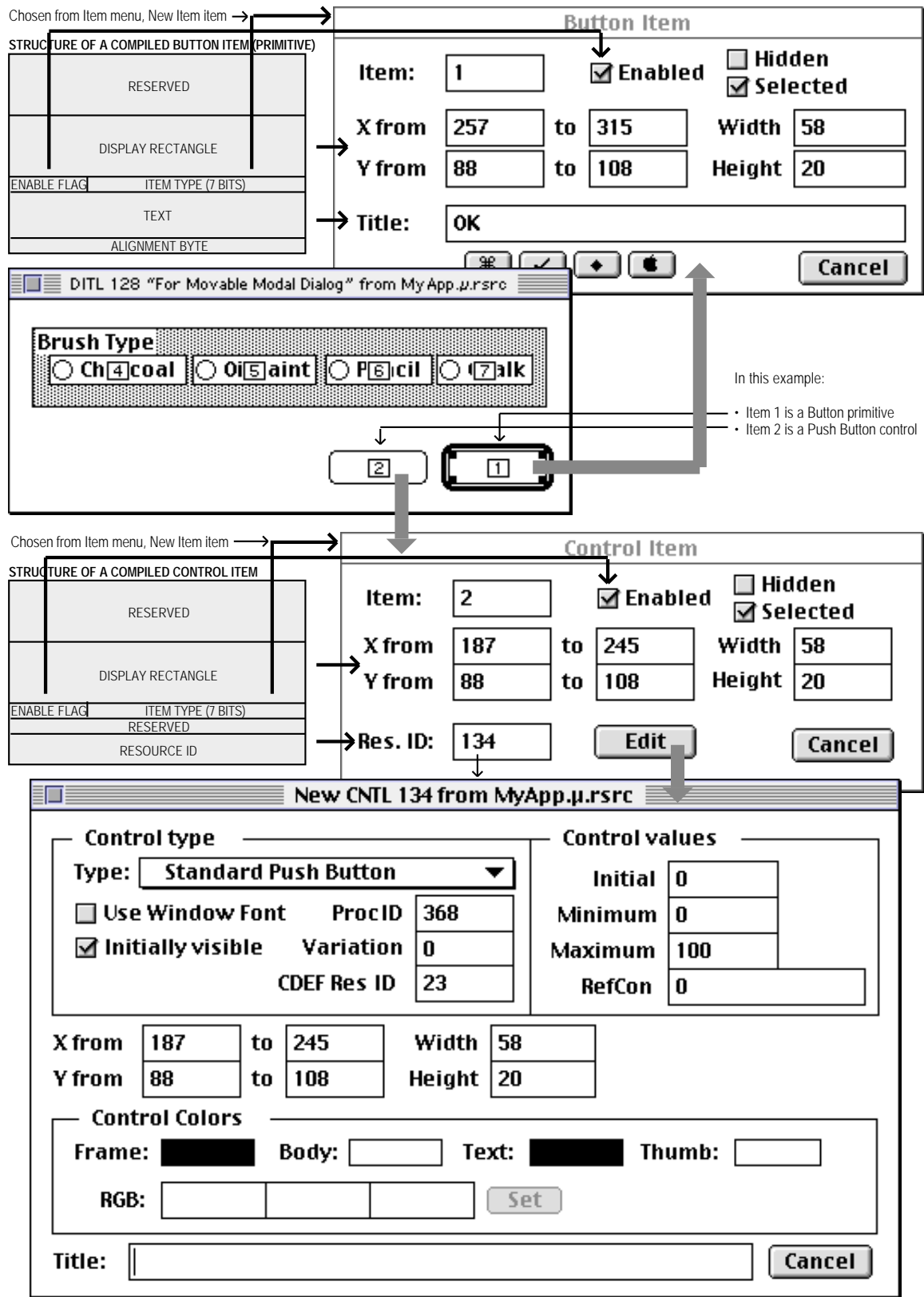


FIG 13 - CREATING A 'DITL' RESOURCE USING RESORCERER

Layout Guidelines For Alert and Dialog Boxes

Layout guidelines for items in alert and dialog boxes are contained in the Apple publication Mac OS 8 Human Interface Guidelines. These guidelines show, amongst other things, the required spacing between items.

Default Push buttons

Default Push button in Alert Boxes

The first item in the item list for an alert box created using an 'ALRT' resource should always be the **OK** push button, because the Dialog Manager will then automatically draw the default ring around that push button. This applies whether the item is a button primitive or a push button control. If a **Cancel** push button is required, it should be the second item.

Default Push button in Dialog Boxes

You should give every dialog a default push button, except for those which contain edit text fields that accept Return key presses. If you do not provide an event filter function (see Event Filter Functions For Modal and Movable Modal Alert and Dialog Boxes, below) which specifies otherwise, the Dialog Manager treats the first item in the item list resource as the default push button for the purpose of responding to Return and Enter key presses.

The Dialog Manager does not automatically draw the default ring around the default push button in dialog boxes.

Enabling and Disabling Items

You should not necessarily enable all items. For example, you typically disable static text field items and edit text field items because your application does not need to respond to clicks in those items.

Do not confuse a *disabled item* with a *deactivated control*. When a control is deactivated, the Control Manager dims it to show that it is deactivated. The Dialog Manager makes no visual distinction between a disabled and enabled item; it simply does not inform your application when the user clicks a disabled item.

Keyboard Focus

Edit text fields and clock controls accept input from the keyboard, and list boxes respond to certain key presses. The Dialog Manager automatically responds to mouse-down events and Tab key-down events intended to shift the keyboard focus between such items, indicating the current target by drawing a keyboard focus frame around that item. For edit text fields, the Dialog Manager also automatically displays the insertion point caret in the current target. For clock controls, the Dialog Manager, in addition to drawing the keyboard focus frame, also moves the keyboard target within the clock by highlighting the individual parts.

The Tab key moves the keyboard focus between such items in a sequence determined by their order in the item list. Accordingly, you should ensure that the item numbers of these items in the 'DITL' resource reflect the sequence in which you require them to be selected by successive Tab key presses.

Manipulating Items

Functions for Manipulating Items

Dialog Manager functions for manipulating items are as follows. Those introduced with Mac OS 8 and the Appearance Manager are shown against a light gray background.

<i>Function</i>	<i>Description</i>
GetDialogItemAsControl	Returns the control handle for an item in an embedding hierarchy. Should be used instead of GetDialogItem (see below) when an embedding hierarchy is established.
GetDialogItem	Returns the control handle (or, for application-defined draw functions, the function pointer), item type, and display rectangle of a given item. When an embedding hierarchy is present, you should generally use GetDialogItemAsControl instead of GetDialogItem to get a handle to the control. When called on a static text field item, GetDialogItem returns a handle to the text, not to the control, and thus may still be used to get a handle to the text of static text fields. (When called on a static text field item, GetDialogItemAsControl returns a handle to the control, not to the text.)
SetDialogItem	When an embedding hierarchy does not exist, sets the item type and the display rectangle of an item or, for application-defined items, the draw function of an item. When an embedding hierarchy exists, you cannot change the type or handle of an item, but application-defined item drawing functions can still be set.
HideDialogItem	Hides the given item.
ShowDialogItem	Re-displays a hidden item.
GetDialogItemText	Returns the text of an edit or static text field item.
SelectDialogItemText	Selects the text of an edit text field item. When embedding is on, you should pass in the control handle produced by a call to GetDialogItemAsControl. When embedding is not on, you should pass in the handle produced by a call to GetDialogItem.
FindDialogItem	Determines the item number of an item at a particular location in a dialog box.
MoveDialogItem	Moves a dialog item to a specified location in a window. Ensures that, if the item is a control, the control rectangle and the dialog item rectangle (maintained by the Dialog Manager) are always the same.
SizeDialogItem	Resizes a dialog item to a specified size. If the dialog item is a control, the control rectangle and the dialog item rectangle (maintained by the Dialog Manager) are always the same.
CountDITL	Counts items in a dialog box.
AppendDITL	Adds items to a dialog box. (See Append Method Constants, below.)
ShortenDITL	Removes items from a dialog box.
ParamText	Substitutes up to four different text strings in static text field items.
GetAlertStage	Returns the stage of the last occurrence of an alert.
ResetAlertStage	Resets the stage of the last occurrence of an alert.

Append Method Constants

The AppendDITL and ShortenDITL functions 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 the following constants in the AppendDITL call:

<i>Constant</i>	<i>Value</i>	<i>Description</i>
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 and Setting The Text in EditText Field and Static Text Field Items

Dialog Manager functions for getting text from, and setting the text of, edit text field and static text field items are as follows:

<i>Function</i>	<i>Description</i>
<code>GetDialogItemText</code>	Gets a copy of the text in static text field and edit text field items. Pass in the handle produced by a call to <code>GetDialogItem</code> , which gets a handle the text in this instance, not to the control.
<code>SetDialogItemText</code>	Sets the text string for static text and edit text fields. When embedding is on, you should pass in the control handle produced by a call to <code>GetDialogItemAsControl</code> . If embedding is not on, pass in the handle produced by <code>GetDialogItem</code> .

The function `ParamText` may also be used to set the text string in a static text field 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 static text field 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.

Setting the Font For Controls in an Alert or Dialog Box — 'dftb' Resources

When an embedding hierarchy is established in a dialog box, you can specify the initial font settings for all controls in an alert or dialog box by creating a **dialog font table resource** (resource type 'dftb') with the same resource ID as the alert or dialog's 'DITL' resource. When a 'dftb' resource is read in, the control font styles are set, and the resource is marked purgeable.

The 'dftb' resource is the resource-based equivalent of the programmatic method of setting a control's font using the function `SetControlFontStyle` described at Chapter 7 — Introduction To Controls.

Structure of a Compiled 'dftb' Resource

Fig 14 shows the structure of a compiled 'dftb' resource and of a constituent dialog font table entry.

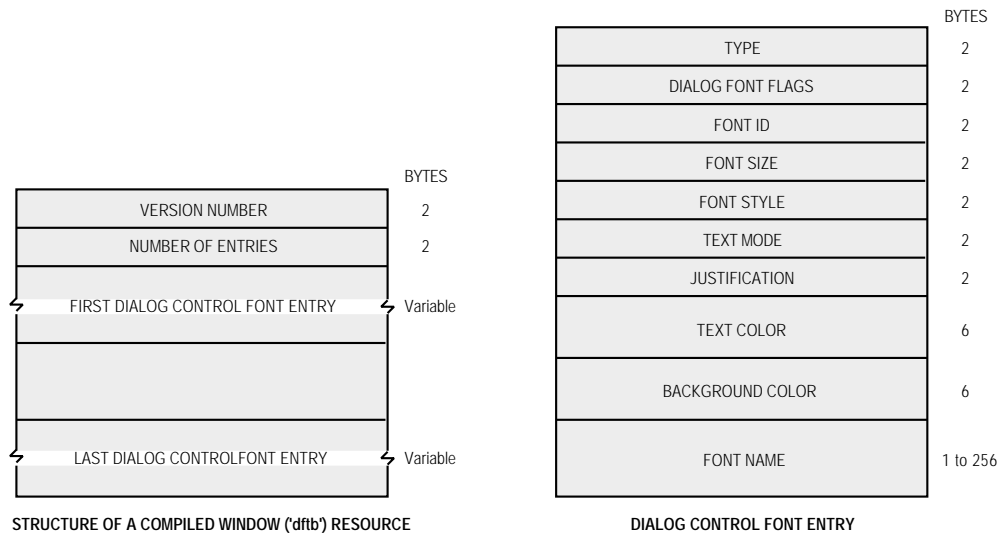


FIG 14- STRUCTURE OF A COMPILED DIALOG FONT TABLE (dfwb) RESOURCE AND A DIALOG CONTROL FONT ENTRY

The following describes the main fields of the 'dfwb' resource and the dialog control font entry:

<i>Field</i>	<i>Description</i>
NUMBER OF ENTRIES	An integer that specifies the number of entries in the resource. Each entry is a dialog control font structure.
FIRST DIALOG CONTROL FONT ENTRY... LAST DIALOG CONTROL FONT ENTRY	A series of dialog control font structures, each of which consist of type, dialog font flags, the font ID, font size, font style, text mode, justification, text color, background color, and font name.
TYPE	An integer that specifies whether there is font information for the dialog or alert item in the 'DITL'. If you specify a value of 0, there is no font information for the item in the corresponding 'DITL', and no data follows. If you specify a value of 1, there is font information for the item, and the rest of the structure is read. You can cause entries to be skipped by setting this value to 0.
DIALOG FONT FLAGS	You can use one or more of these flag constants to specify which other fields in the dialog font table should be used. (See Dialog Font Flag Constants, below.)
FONT ID	If the <code>kDialogFontUseFontMask</code> bit is set to 1, then this element will contain an integer indicating the ID of the font family to use. (See Meta Font Constants, below, for more information about the constants that you can specify.) If this bit is set to 0, then the system default font is used.
FONT SIZE	If a constant representing the system font, small system font, or small emphasized system font is specified in the Font ID field, this field is ignored. If the <code>kDialogFontUseSizeMask</code> bit is set, this field should contain an integer representing the point size of the text. If the <code>kDialogFontAddSizeMask</code> bit is set, this value will contain the size to add to the current point size of the text.
STYLE	If the <code>kDialogFontUseFaceMask</code> bit is set, then this element should contain an integer specifying the text style to describe which styles to apply to the text. You can use one or more of the following style data type mask constants to specify font style: Bit Value Style 0x00 Normal 0x01 Bold 0x02 Italic 0x04 Underline 0x08 Outline 0x10 Shadow 0x20 Condense 0x40 Extend
TEXT MODE	If the <code>kDialogFontUseModeMask</code> bit is set, then this element should contain an integer specifying how characters are drawn. (See Chapter 12 — Drawing With QuickDraw for a discussion of transfer modes.)
JUSTIFICATION	If the <code>kDialogFontUseJustMask</code> bit is set, then this element should contain an integer specifying text justification (left, right, centered, or system-justified).

TEXT COLOR	If the <code>kDialogUseFontForeColorMask</code> bit is set, then this element should contain a colour to use when drawing the text.
BACKGROUND COLOR	If the <code>kDialogFontUseBackColorMask</code> bit is set, then this element should contain a colour to use when drawing the background behind the text. In certain text modes, background colour is ignored.
FONT NAME	If the <code>kDialogFontUseFontNameMask</code> bit is set, then this element should contain a Pascal string representing the font name to be used. This overrides the font ID.

Dialog Font Flag Constants

You can set the following bits in the the dialog font flags field of a dialog control font entry to specify the fields in the entry that should be used

<i>Constant</i>	<i>Value</i>	<i>Meaning</i>
<code>kDialogFontNoFontStyle</code>	<code>0x0000</code>	No font style information is applied.
<code>kDialogFontUseFontMask</code>	<code>0x0001</code>	The font ID specified in the Font ID field of the dialog font table is applied.
<code>kDialogFontUseFaceMask</code>	<code>0x0002</code>	The font style specified in the Style field of the dialog font table is applied.
<code>kDialogFontUseSizeMask</code>	<code>0x0004</code>	The font size specified in the Font Size field of the dialog font table is applied.
<code>kDialogFontUseForeColorMask</code>	<code>0x0008</code>	The text color specified in the Text Color field of the dialog font table is applied. This flag only applies to static text controls.
<code>kDialogFontUseBackColorMask</code>	<code>0x0010</code>	The background color specified in the Background Color field of the dialog font table is applied. This flag only applies to static text controls.
<code>kDialogFontUseModeMask</code>	<code>0x0020</code>	The text mode specified in the Text Mode field of the dialog font table is applied.
<code>kDialogFontUseJustMask</code>	<code>0x0040</code>	The text justification specified in the Justification field of the dialog font table is applied.
<code>kDialogFontUseAllMask</code>	<code>0x00FF</code>	All flags in this mask will be set except <code>kDialogFontAddFontSizeMask</code> and <code>kDialogFontUseFontNameMask</code> .
<code>kDialogFontAddFontSizeMask</code>	<code>0x0100</code>	The Dialog Manager will add a specified font size to the existing font size indicated in the Font Size field of the dialog font table resource.
<code>kDialogFontUseFontNameMask</code>	<code>0x0200</code>	The Dialog Manager will use the string in the Font Name field for the font name instead of a font ID.

Meta Font Constants

You can use the following meta font constants in the font ID field of a dialog control font entry to specify to specify the style, size, and font family of a control's font. You should use these meta font constants whenever possible because the system font can change, depending upon the current theme. If none of these constants are specified, the control uses the system font unless a control with a variant that uses the window font has been specified.

<i>Constant</i>	<i>Value</i>	<i>Meaning In Roman Script System</i>
<code>kControlFontBigSystemFont</code>	<code>0x0001</code>	Use the system font. (For the Roman script system, this is Charcoal 12.)
<code>kControlFontSmallSystemFont</code>	<code>0x0002</code>	Use the small system font. (For the Roman script system, this is Geneva 10.)
<code>kControlFontSmallBoldSystemFont</code>	<code>0x0004</code>	Use the small emphasized system font. (For the Roman script system, this is Geneva 10.)

Another advantage of using these meta font constants is that you can be sure of getting the correct font on a Macintosh using a different script system, such as kanji.

Historical Note

Prior to the introduction of Mac OS 8 and the Appearance Manager, icon colour table ('ictb') resources were used to specify the colour, typeface, font style, and font size for items in an alert or dialog box. When an embedding hierarchy is present, the 'ictb' resource is ignored.

Creating a 'dfbt' Resource Using Resorcerer

Fig 15 shows a dialog control font entry in a 'dfbt' resource being edited with Resorcerer.

For item 6, 0 has been specified as the type, meaning that no data follows. This causes the entry to be skipped.

For item 7, 1 has been specified as the type, meaning that data follows.

kDialogFontUseFontMask bit set, meaning that the font ID specified at Font name is applied.

kDialogFontUseJustMask bit is set, meaning that the text justification specified at Justification is applied.

kDialogFontUseForeColorMask bit is set, meaning that the text colour specified at Foreground color field is applied.

In this example, Helvetica 10 pt font has been specified. However, meta fonts constants could have been specified at this pop-up

TYPE
DIALOG FONT FLAGS
FONT ID
FONT SIZE
FONT STYLE
TEXT MODE
JUSTIFICATION
TEXT COLOR
BACKGROUND COLOR
FONT NAME

For an explanation of r,g,b colour, see Chapter 13 — Color QuickDraw

FIG 15 - CREATING A 'dfbt' RESOURCE USING RESORCERER

Displaying Alert and Dialog Boxes

As previously stated:

- Alert, NoteAlert, CautionAlert and StopAlert are used to create and display alert boxes using descriptive information supplied by 'ALRT' and 'alrx' resources, and StandardAlert is used, in conjunction with information supplied by a standard alert structure, to create and display alert boxes programmatically.

- `GetNewDialog` is used to create dialog boxes using descriptive information supplied by 'DLOG' and 'dlgx' resources, and `NewFeaturesDialog` is used to create dialog boxes programmatically. Both creation methods allow you to specify whether the dialog box is to be initially visible, and both allow you to specify whether or not the dialog box is to be brought to the front of all other windows when it is opened.

To display a dialog box which is specified to be invisible on creation, you must call `ShowWindow` following the `GetNewDialog` or `NewFeaturesDialog` call to display the dialog box. In addition, you should invariably pass `(WindowPtr) -1` in the `behind` and `inBehind` parameters of, respectively, `GetNewDialog` and `NewFeaturesDialog` call so as to display a dialog box as the active (frontmost) window.

Window Deactivation and Menu Adjustment

When an alert box or dialog box is displayed:

- The frontmost window (assuming one exists) must be deactivated.
- The application's menus must be adjusted to reflect the differing levels of permitted menu access which apply in the presence of the various types of alert box and dialog box. (As will be seen, the system software automatically performs some of this menu adjustment for you.)

Historical Note

Prior to the introduction of Mac OS 8 and the Appearance Manager, window deactivation when a movable modal dialog box was displayed was handled in the same way as applies in the case of a modeless dialog box, that is, within the application's main event loop. However, under Appearance, when the `kDialogFlagsHandleMovableModal` bit is set in the 'dlgx' resource, or in the `inFlags` parameter of `NewFeaturesDialog`, `ModalDialog` is used to handle all user interaction within the dialog. (Previously, this user interaction was handled within the main event loop.) This has implications for the way your application deactivates the front window when a movable modal dialog is displayed.

Prior to the introduction of Mac OS 8 and the Appearance Manager, menu adjustment when a movable modal dialog box was displayed was performed by the application. However, when a movable modal dialog is created by setting the `kDialogFlagsHandleMovableModal` bit in the 'dlgx' resource, or in the `inFlags` parameter of `NewFeaturesDialog`, menu adjustment is performed by the Dialog Manager and Menu Manager.

All that follows assumes that the `kDialogFlagsHandleMovableModal` bit is set in the 'dlgx' resource, or in the `inFlags` parameter of `NewFeaturesDialog`, and that, as a consequence:

- Your application calls `ModalDialog` to handle all user interaction within movable modal dialog boxes (as is the case with modal dialog boxes).
- Menu adjustment will be performed automatically by the Dialog Manager and Menu Manager when a movable modal dialog is displayed (as is the case with modal dialog boxes).

Window Deactivation — Modeless Dialog Boxes

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

Window Deactivation — Modal and Movable Modal Alert and Dialog Boxes

When a modal or movable modal dialog box is created and displayed, your application calls `ModalDialog` to handle all user interaction within the dialog until the dialog is dismissed. Events, which are ordinarily handled within your application's main event loop, will then be trapped and handled by `ModalDialog`. This means that your window activation/deactivation function will not now be called as it normally would following the opening of a new window. Accordingly, if one of your application's

windows is active, you must *explicitly* deactivate it before displaying a modal or movable modal alert or dialog box.

If your application does not display an alert box during certain alert stages, you should call the `GetAlertStage` function to test for those stages before deactivating the active window.

Menu Adjustment — Modeless Dialog Boxes

When your application displays a modeless dialog box, it is responsible for all menu disabling and enabling. Your application should thus perform the following tasks:

- Disable only those menus whose items are invalid in the current context.
- If the modeless dialog box includes edit text fields, enable the **Edit** menu and use the Dialog Manager functions `DialogCut`, `DialogCopy`, `DialogPaste` and `DialogDelete` to support the **Cut**, **Copy**, **Paste**, and **Clear** items in that menu.

Your application is also responsible for all menu enabling when a modeless dialog box is dismissed.

Menu Adjustment — Modal Alert and Dialog Boxes

When your application displays a modal alert or dialog box, the Dialog Manager and Menu Manager interact to provide varying degrees of access to the menus in your menu bar, as follows:

- System software disables the **Application** menu, and all items in the **Help** menu except the **Show Balloons/Hide Balloons** item.
- 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 edit text field, and if the menu bar contains a menu having items with the standard keyboard equivalents for **Cut**, **Copy** and **Paste**, the Menu Manager enables that menu and those three items.

Modal Dialog Boxes with Edit Text Field Items

When your application displays modal dialog boxes with no edit text field items, it can safely allow system software to handle menu bar access. However, because system software cannot handle the **Undo** and **Clear** commands (or any other context-dependent command), you may wish to handle your own menu bar access for modal dialog boxes with edit text field items by performing the following tasks:

- Disable the **Apple** menu or its first item (typically, the **About...** item) 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 items in the **Edit** menu.
- Use `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the **Cut**, **Copy**, **Paste**, and **Clear** items in edit text fields.
- Provide your own code for supporting the **Undo** item.

- Enable your application's items in the **Help** menu as appropriate.

When the user dismisses the modal alert or 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.

Menu Adjustment — Movable Modal Alert and Dialog Boxes

When your application displays a movable modal alert or dialog box, the Dialog Manager and Menu Manager interact to provide the same access to the menus in your menu bar as applies in the case of modal alert and dialog boxes except that, in this case, the **Help** and **Application** menus are enabled.

The alternative of taking control of menu bar access from the Dialog Manager where the dialog box contain edit text field items also applies in the case of movable modal dialogs.

When the user dismisses the movable modal alert or 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.

Displaying Multiple Alert and Dialog Boxes

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

Resizing a Dialog

You can use the function `AutoSizeDialog` to automatically resize static text fields and their dialog boxes to accommodate changed static text. For each static text field item found, `AutoSizeDialog` adjusts the static text field and the bottom of the dialog box window. Any items below a static text field are moved down. If the dialog box is visible, when `AutoSizeDialog` is called, it is hidden, resized, and then shown.

Displaying Alert and Dialog Boxes From the Background

If you ever need to display a modal alert or 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 23 — Miscellany for a description of the Notification Manager).

Handling Events in Alert and Dialog Boxes

Overview

Modal and Movable Modal Alert and Dialog Boxes

When `Alert`, `NoteAlert`, `CautionAlert`, `StopAlert`, and `StandardAlert` are used to display alert boxes, the Dialog Manager handles all of the events generated by the user until the user clicks a push button. (These functions are actually just variations of `ModalDialog` which create and destroy their own windows.) When the user hits a push button, these functions highlight the push button briefly, close the alert box and report the user's selection to the application.

As previously stated, `ModalDialog` handles all user interaction within modal and movable modal dialogs. When the user selects an enabled item, `ModalDialog` reports that the user selected the item and then exits. Your application is then responsible for performing the appropriate action in relation to that item. Your application typically calls `ModalDialog` repeatedly until the user dismisses the dialog.

The `modalFilter` parameters in the `Alert`, `NoteAlert`, `CautionAlert`, `StopAlert`, `StandardAlert` and `ModalDialog` functions, and the `filterProc` field of the standard alert structure associated with the `StandardAlert` function, take a universal procedure pointer to an **event filter function**. The Dialog Manager provides a **standard event filter function**, which is used if `NULL` is passed in the `modalFilter` parameter or `filterProc` field; however, you should supply an application-defined event filter function for modal and movable modal alert and dialog boxes so as to avoid a basic limitation of the standard event filter function. (See *Event Filter Functions For Modal and Movable Modal Alert and Dialog Boxes*, below)

Modeless Dialog Boxes

For modeless dialog boxes, you can use the function `IsDialogEvent` to determine whether the event occurred while a modeless dialog box was the frontmost window and then, optionally, use the function `DialogSelect` to handle the event if it belongs to a modeless dialog box. `DialogSelect` is similar to `ModalDialog` except that it returns control after every event, not just events relating to an enabled item. Also, `DialogSelect` does not pass events to an event filter function.

Responding to Events in Controls

Controls and Control Values

For clicks in those types of controls for which you need to determine or change the control's value, your application should use the Control Manager functions `GetControlValue` and `SetControlValue` to get and set the value. When the user clicks on the **OK** push button, your application should perform whatever action is necessary according to the values returned by the controls.

Controls That Accept Keyboard Input

Edit text fields and clock controls, which both accept keyboard input, 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 in the edit text field control, or the clock's date/time value, when the user clicks the **OK** push button.

When you use `ModalDialog` (key-down events in edit text fields and clock controls in modal or movable modal dialogs) or `DialogSelect` (key-down events in edit text fields and clock controls in modeless dialogs), keystrokes and mouse actions within those controls are handled automatically. In the case of an edit text field, this means that:

- When the user clicks the item, a blinking vertical bar, called the **insertion point 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 and keyboard focus frame automatically advances to the next edit text field, clock control, or list box (if any) in the item list, wrapping around to the first one if there are no more items.

Caret Blinking in Edit Text Fields

`ModalDialog` will cause the the insertion point caret to blink in edit text fields in modal and movable modal dialog boxes. For edit text fields in a modeless dialog box, you should call `IdleControls` in your

main event loop's idle processing function. `IdleControls` calls the edit text field control with an idle event so that the control can call `TEIdle` to make the insertion point caret blink.⁵

Historical Note

`IdleControl`, and idle processing within certain control definition functions, was introduced with Mac OS 8 and the Appearance Manager. Prior to the introduction of Mac OS 8 and the Appearance Manager, the method used to cause the insertion point caret to blink in a modeless dialog was to call `DialogSelect` in the main event loop's idle processing function. `DialogSelect` calls `TEIdle` to make the caret blink. This method remains a valid alternative.

Responding to Events in Modal and Movable Modal Alert Boxes

After displaying a modal or movable modal alert box, `Alert`, `NoteAlert`, `CautionAlert`, `StopAlert`, and `StandardAlert` call `ModalDialog` to handle events automatically.

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

If the event is a mouse-down outside the content region of a movable modal alert box and within a window belonging to the application, `ModalDialog` emits the system alert sound and gets the next event. If the mouse-down is not within the content region or a window belonging to the application, `ModalDialog` performs alert box dragging (if the mouse-down is within the title bar) or sends the application to the background (if the mouse-down is not within the title bar).

`ModalDialog` is continually called until the user selects an enabled control, at which time `Alert`, `NoteAlert`, `CautionAlert`, `StopAlert`, and `StandardAlert` 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 push button. When you write your own event filter function, you should ensure that that function retains this behaviour. 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, `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 the alert box window.
- If the user presses the mouse button while the cursor is in a trackable 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 a disabled item, or if it is in no item, or if any other event occurs, nothing happens.

Responding To Events in Modal and Movable Modal Dialog Boxes

Your application should call `ModalDialog` immediately after displaying a modal or movable modal dialog box. `ModalDialog` repeatedly handles events inside the dialog box until an event involving an enabled item occurs, at which time `ModalDialog` exits, returning the item number. Your application

⁵ You should also ensure that, when caret blinking is required, the `sleep` parameter in the `WaitNextEvent` call is set to a value no greater than that returned by `GetCaretTime`.

should then respond appropriately. Your application should continually call `ModalDialog` until the user clicks on the **OK** or **Cancel** push button, at which time your application should close the dialog box.

If the event is a mouse-down anywhere outside the content region of a modal dialog box, `ModalDialog` emits the system alert sound and gets the next event.

If the event is a mouse-down outside the content region of a movable modal dialog box and within a window belonging to the application, `ModalDialog` emits the system alert sound and gets the next event. If the mouse down is not within the content region or a window belonging to the application, `ModalDialog` performs dialog box dragging (if the mouse-down is within the title bar) or sends the application to the background (if the mouse-down is not within the title bar).

If your event filter function does not handle the event, `ModalDialog` handles the event as follows:

- If the event is an activate or update event for the dialog box, `ModalDialog` activates or updates the dialog box window.
- If the event is a mouse-down while the cursor is in a control that accepts keyboard input (that is, an edit text field or a clock control), `ModalDialog` responds to the mouse activity as appropriate, that is, by either displaying an insertion point or by selecting text in an edit text field or by highlighting the appropriate part of the clock control. Where there is more than one control that accepts keyboard input, `ModalDialog` moves the keyboard focus to that control. If a key-down event occurs and there is an edit text field in the dialog, `ModalDialog` uses `TextEdit` to handle text entry and editing automatically. If the edit text field is enabled, `ModalDialog` returns its item number after it receives either the mouse-down or key-down event. (Normally, edit text fields are disabled, and you use the `GetDialogItemText` function to read the information in the items only after the user clicks the **OK** button.)
- If the event is a mouse-down while the cursor is in a trackable control, `ModalDialog` calls the Control Manager function `TrackControl`. If the user releases the mouse button while the cursor is in an enabled control, `ModalDialog` returns the control's item number. Your application should then respond appropriately.
- If the event is a Tab key key-down event and there is more than one control that accepts keyboard input, `ModalDialog` moves the keyboard focus to the next such item in the item list.
- If the event is a mouse-down while the cursor is in a disabled item or in no item, or if any other event occurs, `ModalDialog` does nothing.

Specifying the Events To Be Received by ModalDialog

The function `SetModalDialogEventMask` may be used to specify the events to be received by the `ModalDialog` function for a given modal or movable modal dialog box. This allows your application to specify additional events that are not by default received by `ModalDialog`, such as disk-inserted events and operating system events. If you use this function to change the `ModalDialog` function's event mask, you must pass `ModalDialog` a pointer to your own event filter function to handle the added events.

You can ascertain the events to be received by `ModalDialog` by calling `GetModalDialogEventMask`.

Historical Note

`SetModalDialogEventMask` and `GetDialogEventMask` were introduced with Mac OS 8.5.

Simulating Item Selection

You can cause the Dialog Manager to simulate item selection in a modal or movable modal dialog box using the function `SetDialogTimeout`. Your application calls this function each time you wish to start a countdown for a specified duration for a specified dialog box. When the specified time elapses, the

Dialog Manager simulates a click on the button specified in the `SetDialogTimeout` call. The Dialog Manager will not simulate item selection until `ModalDialog` processes an event.

You can ascertain the original countdown duration, the time remaining, and the item selection to be simulated by calling `GetDialogTimeout`.

Historical Note

`SetDialogTimeout` and `GetDialogTimeout` were introduced with Mac OS 8.5.

Event Filter Functions For Modal and Movable Modal Alert and Dialog Boxes

In early versions of the system software, when a single application controlled the computer, the standard event filter for the (modal) alert and modal dialog boxes of the day was usually sufficient. However, because the standard event filter does not cater for the updating of either the parent application's windows or those belonging to background applications, it has long been inadequate. Your application should therefore provide an event filter function which compensates for this inadequacy. In most cases, you can use the same filter function for all of your modal and movable modal alert and dialog boxes.

The standard event filter function performs the following checks and actions:

- Checks whether the user has pressed the Return or Enter key and, if so, highlights the default push button for eight ticks and returns the item number of that push button.⁶
- For dialog boxes only, and only if the application has previously called certain Dialog Manager functions (see below):
 - Checks whether the user has pressed the Escape key or Command-period and, if so, highlights the Cancel push button for eight ticks and returns the item number of that button.
 - Check whether the cursor is over an edit text field and, if so, changes the cursor shape to the I-Beam cursor.

As a minimum, your application-defined event filter function should ensure that these checks and actions are performed and should also:

- Handle update events not belonging to the alert or dialog box so as to allow the application to update its own windows, and return `false`. (Note that, by responding to update events in the application's own windows in this way, you also allow `ModalDialog` to perform a minor switch when necessary so that background applications can update their windows as well.)
- Return `false` for all events that your event filter function does not handle.

Defining an Event Filter Function

Part of the recommended approach to defining a basic event filter function is to continue to use the standard event filter function to perform its checks and actions as described above. This requires certain preliminary action which, for dialog boxes, requires calls similar to the following examples after the dialog is created and before the call to `ModalDialog`:

```
// Tell the Dialog Manager which is the default push button item, alias the Return and
// Enter keys to that item, and draw the default ring around that item.

SetDialogDefaultItem(myDialogPtr,iOK);
```

⁶ Unless informed otherwise, the Dialog Manager assumes that the first item in the item list is the default push button. As will be seen, it is possible to specify another push button as the default push button in dialog boxes; however, changing the default push button in alert boxes is possible only when the alert box is created using `StandardAlert`. This is why the default push button in an 'ALRT' resource must be the first item in the item list.


```

// Tell the Dialog Manager which is the Cancel push button item, and alias the escape
// key and Command-period key presses to that item.

SetDialogCancelItem(myDialogPtr,iCancel);

// Tell the Dialog Manager to track the cursor and change it to the I-Beam cursor shape
// whenever it is over an edit text field.

SetDialogTracksCursor(myDialogPtr,true);

```

Note that, for all this to work, it is essential that default and Cancel push buttons, and edit text fields, be specified as primitives, not as actual controls, in the 'DLOG' resource.

With those preparations made, you would define your basic event filter function as in the following example:

```

pascal Boolean  eventFilter(DialogPtr dialogPtr,EventRecord *eventFuncPtr,
                           SInt16 *itemHit)
{
  Boolean  handledEvent;
  GrafPtr  oldPort;

  handledEvent = false;

  if((eventFuncPtr->what == updateEvt) &&
     ((WindowPtr) eventFuncPtr->message != dialogPtr))
  {
    // If the event is an update event, and if it is not for the dialog or alert, call
    // your application's window updating function, and return false.

    doUpdate(eventFuncPtr);
  }
  else
  {
    // If the event was not an update, first save the current graphics port and set the
    // alert box or dialog box's graphics port as the current graphics port. This is
    // necessary when you have called SetDialogTrackCursor to cause the Dialog Manager
    // to track cursor position.

    GetPort(&oldPort);
    SetPort(dialogPtr);

    // Pass the event to the standard event filter function for handling. If the
    // function handles the event, it will return true and, in the itemHit parameter,
    // the number of the item that it handled. ModalDialog, Alert, NoteAlert,
    // CautionAlert, StopAlert, and StandardAlert then return this item number in their
    // own itemHit parameter.

    handledEvent = StdFilterProc(dialogPtr,eventFuncPtr,itemHit);

    // Make the saved graphics port the current graphics port again.

    SetPort(oldPort);
  }

  // Return true or false, as appropriate.

  return(handledEvent);
}

```

Alert, NoteAlert, CautionAlert, StopAlert, StandardAlert and ModalDialog pass events to your event filter function *before* handling each event⁷, and will handle the event if your event filter function returns false.

You can also use your event filter function to handle events that ModalDialog does not handle, such as disk-inserted events⁸, keyboard equivalents, and mouse-down events.

⁷ A major difference between modal alert and dialog boxes and movable modal alert and dialog boxes is that, in the case of the latter, *all* events are passed to your event filter function for handling. This allows you to, for example, handle suspend and resume events when your application is either moved to the background or brought to the front, as well as other events you might want to handle.

Alert Boxes and the Event Filter Function

The example event filter, as written, has a limitation in the case of alert boxes with **Cancel** buttons. Because the standard event filter function is used, and because the `SetDialogCancelItem` function does not apply in the case of alert boxes, the filter function will not check whether the user has pressed the Escape key or Command-period, highlight the **Cancel** push button, or return the item number of that button. It is possible to write an event filter function which rectifies that deficiency; however, a better approach is to simply use `StandardAlert` to create alert boxes which require **Cancel** buttons. The standard alert structure associated with `StandardAlert` allows you to specify which are to be the default and **Cancel** buttons in an alert. This is, in effect, the alert box equivalent of the `SetDialogDefaultItem` and `SetDialogCancelItem` calls used in the case of dialog boxes.

Responding to Events in Modeless Dialog Boxes

As previously stated, you can use the function `IsDialogEvent` to determine whether an event occurred in a modeless dialog box or a document window and then call `DialogSelect` to handle the event if it occurred in a modeless dialog box. `DialogSelect` handles the event as follows:

- If the event is an activate or update event, `DialogSelect` activates or updates the modeless dialog box and returns `false`.
- If the event is a key-down or auto-key event, and there is an edit text field in the modeless dialog box, `DialogSelect` uses `TextEdit` to handle text entry and editing and returns `true` and the item number. If there is no edit-text field, `DialogSelect` returns `false`.
- If the user presses the mouse button while the cursor is in an edit text field, `DialogSelect` responds to the mouse activity as appropriate, that is, by either displaying the insertion point caret or selecting text. `DialogSelect` returns `false` if the edit text field is disabled, and `true` and the item number if it is enabled. (Normally, edit text field items should be disabled.)
- If the user presses the mouse button while the cursor is in an enabled trackable control, `DialogSelect` calls `TrackControl` and, if the user releases the mouse button while the cursor is still within the control, returns `true` and the item number.
- If the user presses the mouse button while the cursor is on a disabled item, or if it is in no item, or if any other event occurs, `DialogSelect` does nothing.

In the case of a key-down or auto-key event in an edit text field, you will ordinarily need to filter out Return and Enter key presses and certain Command-key equivalents so that they are not passed to `DialogSelect`. In the case of Return and Enter key presses, you should also highlight the associated push button for eight ticks before calling the application-defined function which responds to hits on the **OK** button. In the case of Command-key presses, you should only allow Command-X, Command-C, and Command-V to be passed to `DialogSelect` (so that `DialogSelect` can support cut, copy, and paste actions within the edit text field) and pass any other Command-key equivalents to your application's menu handling function.

Closing Dialog Boxes

Use `CloseDialog` to dispose of a dialog box if you allocated the memory for the dialog structure 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, such as control records and text handles. `CloseDialog` does not dispose of the dialog structure or the 'DITL' resource.

⁸ 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.

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

For modeless 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.

Balloon Help For Alert and Dialog Boxes

Two basic options are available for adding help balloons to alert and dialog boxes:

- Adding a **balloon help item** to the item list ('DITL') resource, which will associate either a rectangle help ('hrct') resource or a **dialog help** ('hdlg') resource with that 'DITL' resource. Each hot rectangle component in the 'hrct' resource, and each dialog item component in the 'hdlg' resource, corresponds to an item number in the 'DITL' resource.
- By supplying a window help ('hwin') resource, which will associate help balloons defined in either 'hrct' resources or 'hdlg' resources with the alert or dialog's window.⁹

The option of using a balloon help item (usually referred to as simply a "help item") overcomes the major limitation of the 'hwin' resource methodology, which is the inability to adequately differentiate between alert and dialog boxes with no titles (see Chapter 4 — Windows, Fig 12). On the other hand, adopting the help item methodology means that you can only associate help balloons with items in the 'DITL' resource; you cannot provide a single help balloon for a group of related items (unless, of course, they are grouped within a primary or secondary group box).

Help items are invisible. In Resorcerer, the presence of a balloon help item in a 'DITL' resource is indicated only by a checkmark in the **Balloon Help...** item in the **Item** menu. A help item's presence in the 'DITL' resource is completely ignored by the Dialog Manager.

When the help item methodology is used, the Help Manager automatically tracks the cursor and displays help balloons when the following conditions are met: the alert or dialog box has a help item in its 'DITL' resource; your application calls the Dialog Manager function ModalDialog, IsDialogEvent, Alert, NoteAlert, CautionAlert or StopAlert; balloon help is enabled.

Figs 11 and 12 at Chapter 4 — Windows show 'hrct' and 'hwin' resources being created using Resorcerer.

Fig 16 shows a help item being created using Resorcerer.

⁹ 'hrct' and 'hwin' resources are described at Chapter 4 — Windows.

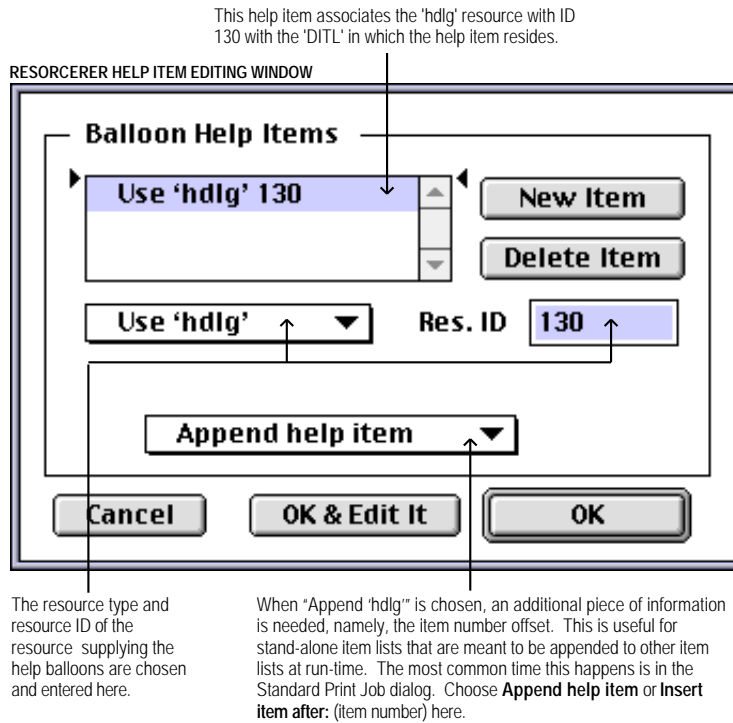
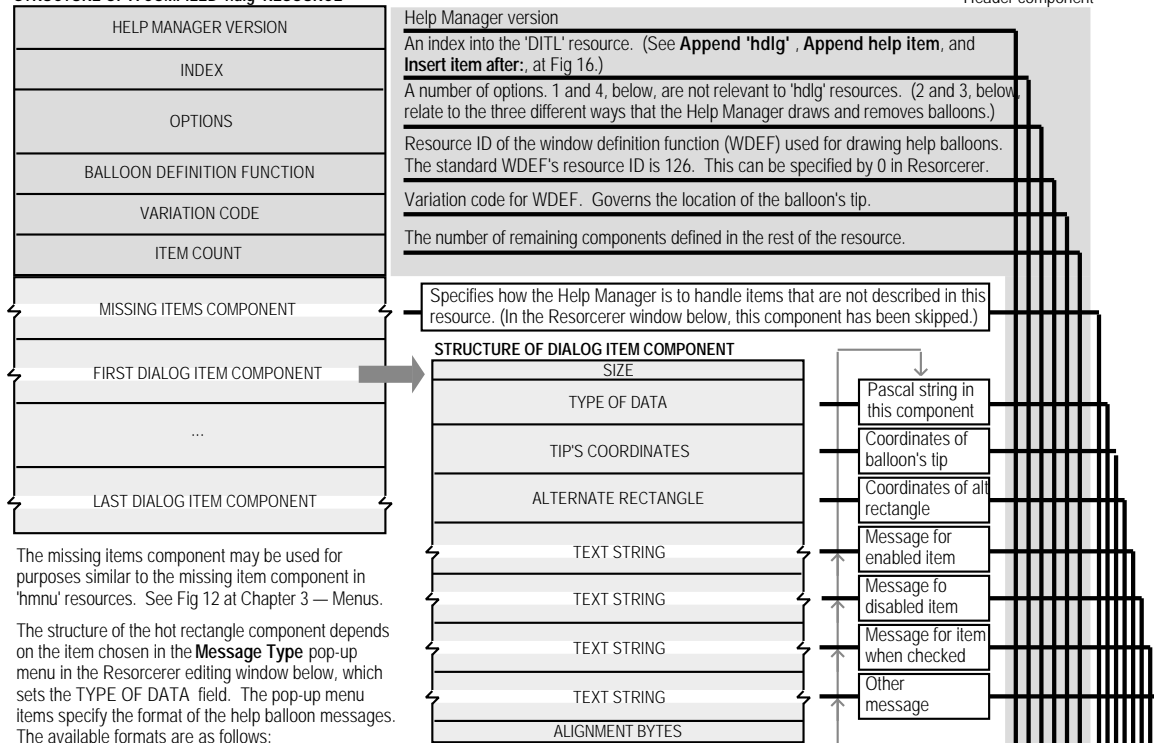


FIG 16 - CREATING A HELP ITEM USING RESORCERER

Fig 17 shows a 'hdlg' (dialog help) resource being created using Resorcerer.

STRUCTURE OF A COMPILED 'hdlg' RESOURCE

Header component



The missing items component may be used for purposes similar to the missing item component in 'hmnul' resources. See Fig 12 at Chapter 3 — Menus.

The structure of the hot rectangle component depends on the item chosen in the **Message Type** pop-up menu in the Resorcerer editing window below, which sets the TYPE OF DATA field. The pop-up menu items specify the format of the help balloon messages. The available formats are as follows:

- Use these strings** Use the string specified within this component of this 'hrcr' resource. (Specified in this example.)
- Use 'PICT' resources** Use the picture stored in the specified 'PICT' resource.
- Use 'STR#' resources** Use the specified text string stored in the specified 'STR#' resource.
- Used styled text resources** Use the styled text stored in the specified 'TEXT' and 'styl' resources.
- Use 'STR' resources** Use the text string stored in the specified 'STR' resource.
- Skip missing item** No help message. Skip this item.

RESORCERER 'hdlg' RESOURCE EDITING WINDOW

hdlg 130 "For Movable Modal Dialog" from MyApp.p.rsrc

Balloon help version Latest=2

Item offset for first message Start messages at Item #1=0

5-31. Reserved 0

4. For 'hwin's, match string anywhere in title Off

3. Create window, restore bits, and cause update Off

2. Don't create window, restore bits, no update Off

1. Pretend window port origin is set to (0,0) Off

0. Treat resource IDs as sub-IDs for owned resources of a D

Balloon 'WDEF' Resource ID Standard balloons=0

Balloon variation code (tip position) Along left side at top=0

Dialog items 4

Number of bytes to next record

Missing message type Skip missing item=256

Dialog items #1

Number of bytes to next record

Message type Use these strings=1

Tip (x,y)=(0,0)

Hot rect (t,l,b,r)=(0,0,0,0)

Enabled message "Click this radio button to set the brush type to water set

Disabled message

Checked message "Indicates that the current brush type is water set

Other message

Align

New **Ed**

This field is misnamed in Resorcerer. The Help Manager uses an item's display rectangle as the hot rectangle for help balloons. The optional alternate rectangle specified here is used by the Help Manager to transpose the tip if the help balloon does not fit the screen. If the alternate is smaller than the hot, you have greater assurance of the balloon fitting onscreen. If the alternate is larger than the hot, you have greater assurance that the balloon will not obscure some important portion within the hot (display) rectangle.

FIG 17 - CREATING A 'hdlg' RESOURCE USING RESORCERER

Main Dialog Manager Constants, Data Types and Functions

In the following:

- The constants, data types, and functions introduced with Mac OS 8 and the Appearance Manager are shown on a light gray background.
- The functions introduced with Mac OS 8.5 are shown on a dark gray background.

Constants

Dialog Item Types

kControlDialogItem	= 4
kButtonDialogItem	= kControlDialogItem 0
kCheckBoxDialogItem	= kControlDialogItem 1
kRadioButtonDialogItem	= kControlDialogItem 2
kResourceControlDialogItem	= kControlDialogItem 3
kStaticTextDialogItem	= 8
kEditTextDialogItem	= 16
kIconDialogItem	= 32
kPictureDialogItem	= 64
kUserDialogItem	= 0
kItemDisableBit	= 128

Item Numbers for OK and Cancel Push Buttons

kStdOkItemIndex	= 1
kStdCancelItemIndex	= 2

Resource IDs of Alert Box Icons

kStopIcon	= 0
kNoteIcon	= 1
kCautionIcon	= 2

Constants Used for the Method Parameter in AppendDITL

overlayDITL	= 0
appendDITLRight	= 1
appendDITLBottom	= 2

Alert Types

kAlertStopAlert	= 0
kAlertNoteAlert	= 1
kAlertCautionAlert	= 2
kAlertPlainAlert	= 3

Alert Button Constants

kAlertStdAlertOKButton	= 1,
kAlertStdAlertCancelButton	= 2,
kAlertStdAlertOtherButton	= 3,
kAlertStdAlertHelpButton	= 4

Alert Default Text Constants

kAlertDefaultOKText	= -1
kAlertDefaultCancelText	= -1
kAlertDefaultOtherText	= -1

Dialog Feature Flag Constants

kDialogFlagsUseThemeBackground	= (1 << 0)
kDialogFlagsUseControlHierarchy	= (1 << 1)
kDialogFlagsHandleMovableModal	= (1 << 2)
kDialogFlagsUseThemeControls	= (1 << 3)

Alert Feature Flag Constants

```
kAlertFlagsUseThemeBackground = (1 << 0)
kAlertFlagsUseControlHierarchy = (1 << 1)
kAlertFlagsAlertIsMovable = (1 << 2)
kAlertFlagsUseThemeControls = (1 << 3)
```

Dialog Font Flag Constants

```
kDialogFontNoFontStyle = 0
kDialogFontUseFontMask = 0x0001
kDialogFontUseFaceMask = 0x0002
kDialogFontUseSizeMask = 0x0004
kDialogFontUseForeColorMask = 0x0008
kDialogFontUseBackColorMask = 0x0010
kDialogFontUseModeMask = 0x0020
kDialogFontUseJustMask = 0x0040
kDialogFontUseAllMask = 0x00FF
kDialogFontAddFontSizeMask = 0x0100
kDialogFontUseFontNameMask = 0x0200
```

Constants Used for *procID* Parameter in *NewDialog* and *NewColorDialog* and *inProcID* Parameter in *NewFeaturesDialog*

```
kWindowDocumentProc 1024 // Modeless dialog box
kWindowPlainDialogProc 1040 // Modal dialog box
kWindowShadowDialogProc 1041 // Modal dialog box
kWindowModalDialogProc 1042 // Modal dialog box
kWindowMovableModalDialogProc 1043 // Movable modal dialog box
kWindowAlertProc 1044 // Modal alert box
kWindowMovableAlertProc 1045 // Movable modal alert box
```

Data Types

```
typedef WindowPtr DialogPtr;
```

Dialog Structure

```
struct DialogRecord
{
    WindowRecord window; // Dialog's window structure.
    Handle items; // Item list resource.
    TEHandle textH; // Current editable text item.
    SInt16 editField; // Editable text item number minus 1.
    SInt16 editOpen; // (Used internally.)
    SInt16 aDefItem; // Default push button item number.
};
typedef struct DialogRecord DialogRecord;
typedef DialogRecord *DialogPeek;
```

Standard Alert Structure

```
struct AlertStdAlertParamRec
{
    Boolean movable; // Make alert movable modal.
    Boolean helpButton; // Is there a help button?
    ModalFilterUPP filterProc; // Event filter.
    StringPtr defaultText; // Text for button in OK position.
    StringPtr cancelText; // Text for button in cancel position.
    StringPtr otherText; // Text for button in left position.
    SInt16 defaultButton; // Which button behaves as the default.
    SInt16 cancelButton; // Which one behaves as cancel (can be 0).
    UInt16 position; // Position (kWindowDefaultPosition in this case equals
    // kWindowAlertPositionParentWindowScreen).
};
typedef struct AlertStdAlertParamRec AlertStdAlertParamRec;
typedef AlertStdAlertParamRec * AlertStdAlertParamPtr;
```

Functions

Initialising the Dialog Manager

```
void InitDialogs(void *ignored);
void ErrorSound(SoundUPP soundProc);
void SetDialogFont(short value);
```

Creating Alerts

```
short Alert(short alertID, ModalFilterUPP modalFilter);
short StopAlert(short alertID, ModalFilterUPP modalFilte);
short NoteAlert(short alertID, ModalFilterUPP modalFilte);
short CautionAlert(short alertID, ModalFilterUPP modalFilte);
OSErr StandardAlert(AlertType inAlertType, StringPtr inError, StringPtr inExplanation,
AlertStdAlertParamPtr inAlertParam, SInt16 *outItemHit);
short GetAlertStage(void);
void ResetAlertStage(void);
```

Creating and Disposing of Dialog Boxes

```
DialogPtr GetNewDialog(short dialogID, void *dStorage, WindowRef behind);
DialogPtr NewDialog(void *wStorage, const Rect *boundsRect, ConstStr255Param title, Boolean
visible, short procID, WindowRef behind, Boolean goAwayFlag, long refCon, Handle
itmLstHndl);
DialogPtr NewColorDialog(void *dStorage, const Rect *boundsRect, ConstStr255Param title,
Boolean visible, short procID, WindowRef behind, Boolean goAwayFlag, long refCon,
Handle items);
DialogPtr NewFeaturesDialog(void *inStorage, const Rect *inBoundsRect,
ConstStr255Param inTitle, Boolean inIsVisible, SInt16 inProcID, WindowPtr inBehind,
Boolean inGoAwayFlag, SInt32 inRefCon, Handle inItemListHandle, UInt32 inFlags);
void CloseDialog(DialogPtr theDialog);
void DisposeDialog(DialogPtr theDialog);
```

Manipulating Items in Alert and Dialog Boxes

```
void GetDialogItem(DialogPtr theDialog, short itemNo, short *itemType, Handle *item,
Rect *box);
void SetDialogItem(DialogPtr theDialog, short itemNo, short itemType, Handle item,
const Rect *box);
OSErr GetDialogItemAsControl(DialogPtr inDialog, SInt16 inItemNo,
ControlHandle *outControl);
OSErr MoveDialogItem(DialogPtr inDialog, SInt16 inItemNo, SInt16 inHoriz, SInt16 inVert);
OSErr SizeDialogItem(DialogPtr inDialog, SInt16 inItemNo, SInt16 inHeight, SInt16 inWidth);
OSErr AutoSizeDialog(DialogPtr inDialog);
void HideDialogItem(DialogPtr theDialog, short itemNo);
void ShowDialogItem(DialogPtr theDialog, short itemNo);
short FindDialogItem(DialogPtr theDialog, Point thePt);
void AppendDITL(DialogPtr theDialog, Handle theHandle, DITLMethod theMethod);
void ShortenDITL(DialogPtr theDialog, short numberItems);
short CountDITL(DialogPtr the Dialog);
```

Handling Text in Alert and Dialog Boxes

```
void ParamText(ConstStr255Param param0, ConstStr255Param param1, ConstStr255Param param2,
ConstStr255Param param3);
void GetDialogItemText(Handle item, Str255 text);
void SetDialogItemText(Handle item, ConstStr255Param text);
void SelectDialogItemText(DialogPtr theDialog, short itemNo, short strtSel, short endSel);
void DialogCut(DialogPtr theDialog);
void DialogPaste(DialogPtr theDialog);
void DialogCopy(DialogPtr theDialog);
void DialogDelete(DialogPtr theDialog);
```

Handling Events in Dialog Boxes

```
void ModalDialog(ModalFilterUPP modalFilter, short *itemHit);
Boolean IsDialogEvent(const EventRecord *theEvent);
Boolean DialogSelect(const EventRecord *theEvent, DialogPtr *theDialog, short *itemHit);
void DrawDialog(DialogPtr theDialog);
void UpdateDialog(DialogPtr theDialog, RgnHandle updateRgn);
OSStatus SetModalDialogEventMask(DialogPtr inDialog, EventMask inMask);
```



```

OSStatus GetModalDialogEventMask(DialogPtr inDialog,EventMask *outMask);
OSStatus SetDialogTimeout(DialogPtr inDialog,SInt16 inButtonToPress,UInt32 inSecondsToWait);
OSStatus GetDialogTimeout(DialogPtr inDialog,SInt16 *outButtonToPress,
    UInt32 *outSecondsToWait,UInt32 *outSecondsRemaining);

```

Creating a Routine Descriptor For an Event Filter Function

```

#define NewModalFilterProc(userRoutine)
    (ModalFilterUPP) NewRoutineDescriptor((ProcPtr)(userRoutine),uppModalFilterProcInfo,
    GetCurrentArchitecture())

```

Demonstration Program

```

// #####
// DialogsAndAlerts.c
// #####
//
// If Mac OS 8.5 is present, this program initially opens a small modal dialog box which
// is automatically closed after 10 seconds, the timeout value having been set by a call
// to SetDialogTimeout. The program then:
//
// • Opens a window with an Appearance-compliant window header for the purposes of
// displaying advisory 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 modal alert box, movable modal alert box,
// modal dialog box, movable modal dialog box, and modeless dialog box via the
// Demonstration menu.
//
// The modal alert box is created using 'ALRT' and 'alrx' resources.
//
// The movable modal alert box is created programmatically using the StandardAlert
// function and a standard alert structure.
//
// The modal dialog box contains three checkboxes in one group box, and two pop-up menu
// buttons in another group box.
//
// The movable modal dialog box contains four radio buttons in one group box, and a clock
// control and edit text field in another group box..
//
// The modeless dialog box contains, amongst other items, an edit text field.
//
// The modal alert box, movable modal alert box, modal dialog box, and movable modal
// dialog box use an application-defined event filter function.
//
// The program utilises the following resources:
//
// • An 'MBAR' resource, and 'MENU' resources for Apple, File, Demonstration, and Help
// pull-down menus, and the pop-up menu buttons (preload, non-purgeable).
//
// • A 'WIND' resource (purgeable) (initially visible).
//
// • An 'ALRT' resource (purgeable), together with an associated 'alrx' resource
// (purgeable), and 'dftb' resource (non-purgeable, but 'dftb' resources are auto-
// matically marked purgeable when read in).
//
// • 'DLOG' resources (purgeable) (initially not visible) and associated 'DITL'
// resources (purgeable), 'dlgx' resources (purgeable), and 'dftb' resources (non-
// purgeable, but 'dftb' resources are automatically marked purgeable when read in).
//
// • 'CNTL' resources for primary group boxes, separator lines, pop-up menu buttons,
// a clock, and an image well (all purgeable).
//
// • A 'STR#' resources (purgeable) containing the label and narrative text for the
// movable modal alert box.
//
// • A 'cicn' resource (purgeable) for the modeless dialog box.
//
// • A 'ppat' resource (purgeable), which is used to colour the content region of the
// document window for update proving purposes.
//
// • 'hdlg' resources (purgeable) containing balloon help information for the modal and
// and movable modal dialog boxes.

```

```

//
// • An 'hrct' resource and associated 'hwin' resource (both purgeable) containing
//   balloon help information for the modeless dialog box.
//
// • A 'SIZE' resource with the acceptSuspendResumeEvents and doesActivateOnFGSwitch,
//   and is32BitCompatible flags set.
//
// #####
// ..... includes
#include <Appearance.h>
#include <ControlDefinitions.h>
#include <Devices.h>
#include <Gestalt.h>
#include <LowMem.h>
#include <Resources.h>
#include <Sound.h>
#include <ToolUtils.h>

// ..... defines
#define rMenuBar          128
#define mApple           128
#define iAbout           1
#define mFile            129
#define iClose           4
#define iQuit            11
#define mEdit            130
#define iCut              3
#define iCopy             4
#define iPaste            5
#define iClear            6
#define mDemonstration   131
#define iModalAlert      1
#define iMovableAlert    2
#define iModalDialog     3
#define iMovableModalDialog 4
#define iModeless        5
#define rWindow           128
#define rAlert            128
#define rModal            129
#define iGridSnap        4
#define iShowGrid        5
#define iShowRulers      6
#define iFont             11
#define iSound            12
#define rMovable          130
#define iCharcoal         7
#define iOilPaint         8
#define iPencil           9
#define iChalk            10
#define iClockOne        12
#define rModeless        131
#define iEditText        2
#define rSplash           132
#define rAlertStrings    128
#define sLabel            1
#define sNarrative        2
#define rPixelPattern    128
#define kSearchModeless  1

#define kReturn           (SInt8) 0x0D
#define kEnter            (SInt8) 0x03
#define kEscape           (SInt8) 0x1B
#define kPeriod           (SInt8) 0x2E

#define MAXLONG           0x7FFFFFFF

// ..... global variables
Ptr          gPreAllocatedBlockPtr;
ModalFilterUPP  eventFilterUPP;
Str255        gCurrentString;
WindowPtr      gWindowPtr;
SInt16        gPixelDepth;

```

```

Boolean      gIsColourDevice      = false;
SInt32       gSleepTime;
Boolean      gDone;
Boolean      gInBackground;
Boolean      gGridSnap            = kControlCheckBoxUncheckedValue;
Boolean      gShowGrid           = kControlCheckBoxUncheckedValue;
Boolean      gShowRule           = kControlCheckBoxUncheckedValue;
SInt16       gBrushType          = iCharcoal;
DialogPtr    gSearchModelessDialogPtr = NULL;

// ..... function prototypes

void main (void);
void doInitManagers (void);
void eventLoop (void);
void doIdle (void);
void doEvents (EventRecord *);
void doMouseDown (EventRecord *);
void doKeyDown (EventRecord *);
void doUpdate (EventRecord *);
void doUpdateDocument (WindowPtr);
void doUpdateModelessDialog (EventRecord *);
void doActivate (EventRecord *);
void doOSEvent (EventRecord *);
void doActivateDocument (WindowPtr, Boolean);
void doActivateModelessDialog (EventRecord *, Boolean);
void doAdjustMenus (void);
void doMenuChoice (SInt32);
void doEditMenu (SInt16);
void doDemonstrationMenu (SInt16);
void doExplicitlyDeactivateDocument (void);
Boolean doMovableModalAlert (void);
Boolean doModalDialog (void);
Boolean doMovableModalDialog (void);
Boolean doCreateOrShowModelessDialog (void);
void doInContent (EventRecord *);
void doButtonHitInSearchModeless (void);
void doHideModelessDialog (WindowPtr);
void doPopupMenuChoice (ControlHandle, SInt16, SInt16);
void doPlaySound (Str255);
void doDrawMessage (WindowPtr, Boolean);
void doCopyPString (Str255, Str255);
void doGetDepthAndDevice (void);

pascal Boolean eventFilter (DialogPtr, EventRecord *, SInt16 *);

// ##### main

void main(void)
{
    SInt32 response;
    DialogPtr modalDialogPtr;
    SInt16 itemHit;
    Handle menubarHdl;
    MenuHandle menuHdl;
    OSErr osError;

    // ..... get nonrelocatable block low in heap for modeless dialog's dialog structure
    if(!(gPreAllocatedBlockPtr = NewPtr(sizeof(DialogRecord))))
        ExitToShell();

    // ..... initialise managers
    doInitManagers();

    // ..... open small modal dialog and automatically dismiss it after 10 seconds

#ifdef TARGET_CPU_PPC
    osError = Gestalt(gestaltSystemVersion, &response);

    if(osError == noErr && response >= 0x00000850)
    {
        modalDialogPtr = GetNewDialog(rSplash, NULL, (WindowPtr) -1);

        SetDialogTimeout(modalDialogPtr, kStdOkItemIndex, 10);
    }
#endif
}

```

```

    do
    {
        ModalDialog(NULL,&itemHit);
    } while(itemHit != kStdOkItemIndex);
    DisposeDialog(modalDialogPtr);
}
#endif

// ..... create routine descriptor for event filter function
eventFilterUPP = NewModalFilterProc((ProcPtr) eventFilter);

// ..... set up menu bar and menus
menubarHdl = GetNewMBar(rMenubar);
if(menubarHdl == NULL)
    ExitToShell();
SetMenuBar(menubarHdl);
DrawMenuBar();

menuHdl = GetMenuHandle(mApple);
if(menuHdl == NULL)
    ExitToShell();
else
    AppendResMenu(menuHdl,'DRVR');

// ..... initial advisory text for window header
doCopyPString("\pBalloon help is available for dialog boxes",gCurrentString);

// ..... open a window, set font size
if(!(gWindowPtr = GetNewCWindow(rWindow,NULL,(WindowPtr)-1)))
    ExitToShell();

SetPort(gWindowPtr);
TextSize(10);

// ..... get pixel depth and whether colour device for certain Appearance functions
doGetDepthAndDevice();

// ..... enter eventLoop
eventLoop();
}

// ##### doInitManagers
void doInitManagers(void)
{
    MaxApplZone();
    MoreMasters();

    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);

    InitCursor();
    FlushEvents(everyEvent,0);

    RegisterAppearanceClient();
}

// ##### eventLoop
void eventLoop(void)
{
    EventRecord    eventStructure;
    Boolean        gotEvent;

```

```

gSleepTime = MAXLONG;
gDone = false;

while(!gDone)
{
    gotEvent = WaitNextEvent(everyEvent,&eventStructure,gSleepTime,NULL);

    if(gotEvent)
        doEvents(&eventStructure);
    else
        doIdle();
}
}

// ##### doIdle

void doIdle(void)
{
    if(FrontWindow() == gSearchModelessDialogPtr)
        IdleControls(gSearchModelessDialogPtr);
}

// ##### doEvents

void doEvents(EventRecord *eventStrucPtr)
{
    switch(eventStrucPtr->what)
    {
        case mouseDown:
            doMouseDown(eventStrucPtr);
            break;

        case keyDown:
        case autoKey:
            doKeyDown(eventStrucPtr);
            break;

        case updateEvt:
            doUpdate(eventStrucPtr);
            break;

        case activateEvt:
            doActivate(eventStrucPtr);
            break;

        case osEvt:
            doOSEvent(eventStrucPtr);
            HiliteMenu(0);
            break;
    }
}

// ##### doMouseDown

void doMouseDown(EventRecord *eventStrucPtr)
{
    WindowPtr windowPtr;
    SInt16 partCode;

    partCode = FindWindow(eventStrucPtr->where,&windowPtr);

    switch(partCode)
    {
        case inMenuBar:
            doAdjustMenus();
            doMenuChoice(MenuSelect(eventStrucPtr->where));
            break;

        case inContent:
            if(windowPtr != FrontWindow())
                SelectWindow(windowPtr);
            else
                doInContent(eventStrucPtr);
            break;

        case inDrag:

```

```

DragWindow(windowPtr, eventStrucPtr->where, &qd.screenBits.bounds);
break;

case inGoAway:
    if(TrackGoAway(windowPtr, eventStrucPtr->where))
    {
        if(((WindowPeek) windowPtr)->windowKind == kDialogWindowKind)
        {
            doHideModelessDialog(windowPtr);
            doCopyPString("\pBalloon help is available for dialog boxes", gCurrentString);
        }
    }
    break;
}
}

// ##### doKeyDown

void doKeyDown(EventRecord *eventStrucPtr)
{
    WindowPtr    windowPtr;
    SInt8        charCode;
    SInt16       whichModeless, itemHit;
    ControlHandle controlHdl;
    UInt32       finalTicks;

    windowPtr = FrontWindow();
    charCode = eventStrucPtr->message & charCodeMask;

    if(!(IsDialogEvent(eventStrucPtr)))
    {
        if((eventStrucPtr->modifiers & cmdKey) != 0)
        {
            doAdjustMenus();
            doMenuChoice(MenuEvent(eventStrucPtr));
        }
    }
    else
    {
        if((whichModeless = ((WindowPeek) windowPtr)->refCon) == kSearchModeless)
        {
            if((charCode == kReturn) || (charCode == kEnter))
            {
                GetDialogItemAsControl(windowPtr, kStdOkItemIndex, &controlHdl);
                HiliteControl(controlHdl, kControlButtonPart);
                Delay(8, &finalTicks);
                HiliteControl(controlHdl, kControlNoPart);
                doButtonHitInSearchModeless();
                return;
            }

            if((eventStrucPtr->modifiers & cmdKey) != 0)
            {
                if(charCode == 'X' || charCode == 'x' || charCode == 'C' || charCode == 'c' ||
                    charCode == 'V' || charCode == 'v')
                {
                    HiliteMenu(mEdit);
                    DialogSelect(eventStrucPtr, &(DialogPtr) windowPtr, &itemHit);
                    Delay(4, &finalTicks);
                    HiliteMenu(0);
                }
                else
                {
                    doAdjustMenus();
                    doMenuChoice(MenuEvent(eventStrucPtr));
                }
                return;
            }

            DialogSelect(eventStrucPtr, &(DialogPtr) windowPtr, &itemHit);
        }
    }
}

// ##### doUpdate

```

```

void doUpdate(EventRecord *eventStrucPtr)
{
    WindowPtr windowPtr;

    windowPtr = (WindowPtr) eventStrucPtr->message;

    if(!(IsDialogEvent(eventStrucPtr)))
        doUpdateDocument(windowPtr);
    else
        doUpdateModelessDialog(eventStrucPtr);
}

// ##### doUpdateDocument
void doUpdateDocument(WindowPtr windowPtr)
{
    GrafPtr      oldPort;
    PixPatHandle pixpatHdl;

    BeginUpdate(windowPtr);

    GetPort(&oldPort);
    SetPort(windowPtr);

    pixpatHdl = GetPixPat(rPixelPattern);
    FillRect(&windowPtr->portRect,pixpatHdl);
    DisposePixPat(pixpatHdl);
    doDrawMessage(windowPtr,windowPtr == FrontWindow() && !gInBackground);

    SetPort(oldPort);

    EndUpdate(windowPtr);
}

// ##### doUpdateModelessDialog
void doUpdateModelessDialog(EventRecord *eventStrucPtr)
{
    WindowPtr windowPtr;
    SInt16    itemHit;

    DialogSelect(eventStrucPtr,&windowPtr,&itemHit);
}

// ##### doActivate
void doActivate(EventRecord *eventStrucPtr)
{
    WindowPtr windowPtr;
    Boolean    becomingActive;

    windowPtr = (WindowPtr) eventStrucPtr->message;
    becomingActive = (eventStrucPtr->modifiers & activeFlag) == activeFlag;

    if(!(IsDialogEvent(eventStrucPtr)))
        doActivateDocument(windowPtr,becomingActive);
    else
        doActivateModelessDialog(eventStrucPtr,becomingActive);
}

// ##### doOSEvent
void doOSEvent(EventRecord *eventStrucPtr)
{
    WindowPtr windowPtr;

    switch((eventStrucPtr->message >> 24) & 0x000000FF)
    {
        case suspendResumeMessage:

            windowPtr = FrontWindow();
            gInBackground = (eventStrucPtr->message & resumeFlag) == 0;

            if(!(IsDialogEvent(eventStrucPtr)))
                doActivateDocument(windowPtr,!gInBackground);
            else

```

```

        doActivateModelessDialog(eventStrucPtr,!gInBackground);
        break;

    case mouseMovedMessage:
        break;
    }
}

// ##### doActivateDocument
void doActivateDocument(WindowPtr windowPtr,Boolean becomingActive)
{
    if(becomingActive)
        doAdjustMenus();

    doDrawMessage(windowPtr,becomingActive);
}

// ##### doActivateModelessDialog
void doActivateModelessDialog(EventRecord *eventStrucPtr,Boolean becomingActive)
{
    SInt16        whichModeless;
    WindowPtr     windowPtr;
    SInt16        itemHit;

    DialogSelect(eventStrucPtr,&windowPtr,&itemHit);

    if(becomingActive)
    {
        doAdjustMenus();
        if((whichModeless = ((WindowPeek) windowPtr)->refCon) == kSearchModeless)
            gSleepTime = LMGetCaretTime();
    }
    else
    {
        if((whichModeless = ((WindowPeek) windowPtr)->refCon) == kSearchModeless)
            gSleepTime = MAXLONG;
    }
}

// ##### doAdjustMenus
void doAdjustMenus(void)
{
    WindowPtr     windowPtr;
    MenuHandle     menuHdl;

    windowPtr = FrontWindow();

    if(((WindowPeek) windowPtr)->windowKind == kApplicationWindowKind)
    {
        menuHdl = GetMenuHandle(mFile);
        EnableItem(menuHdl,0);
        DisableItem(menuHdl,4);
        menuHdl = GetMenuHandle(mEdit);
        DisableItem(menuHdl,0);
        menuHdl = GetMenuHandle(mDemonstration);
        EnableItem(menuHdl,0);
        EnableItem(menuHdl,5);
    }
    else if(((WindowPeek) windowPtr)->windowKind == kDialogWindowKind)
    {
        menuHdl = GetMenuHandle(mFile);
        EnableItem(menuHdl,0);
        EnableItem(menuHdl,4);
        menuHdl = GetMenuHandle(mEdit);
        EnableItem(menuHdl,0);
        menuHdl = GetMenuHandle(mDemonstration);
        EnableItem(menuHdl,0);
        DisableItem(menuHdl,5);
    }
}

DrawMenuBar();
}

```



```

// ##### doMenuChoice

void doMenuChoice(SInt32 menuChoice)
{
    SInt16 menuID, menuItem;
    Str255 itemName;
    SInt16 daDriverRefNum;

    menuID = HiWord(menuChoice);
    menuItem = LoWord(menuChoice);

    if(menuID == 0)
        return;

    switch(menuID)
    {
        case mApple:
            if(menuItem == iAbout)
                SysBeep(10);
            else
            {
                GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
                daDriverRefNum = OpenDeskAcc(itemName);
            }
            break;

        case mFile:
            if(menuItem == iQuit)
                gDone = true;
            else if(menuItem == iClose)
            {
                if(((WindowPeek) FrontWindow())->windowKind == kDialogWindowKind)
                    doHideModelessDialog(FrontWindow());
            }
            break;

        case mEdit:
            doEditMenu(menuItem);
            break;

        case mDemonstration:
            doDemonstrationMenu(menuItem);
            break;
    }

    HiliteMenu(0);
}

// ##### doEditMenu

void doEditMenu(SInt16 menuItem)
{
    WindowPtr windowPtr;
    SInt16 whichModeless;

    windowPtr = FrontWindow();

    if(((WindowPeek) windowPtr)->windowKind == kDialogWindowKind)
    {
        if((whichModeless = ((WindowPeek) windowPtr)->refCon) == kSearchModeless)
        {
            switch(menuItem)
            {
                case iCut:
                    DialogCut((DialogPtr) windowPtr);
                    break;

                case iCopy:
                    DialogCopy((DialogPtr) windowPtr);
                    break;

                case iPaste:
                    DialogPaste((DialogPtr) windowPtr);
                    break;

                case iClear:

```

```

        DialogDelete((DialogPtr) windowPtr);
        break;
    }
}
}

// ##### doDemonstrationMenu

void doDemonstrationMenu(SInt16 menuItem)
{
    switch(menuItem)
    {
        case iModalAlert:
            doExplicitlyDeactivateDocument();
            StopAlert(rAlert,eventFilterUPP);
            break;

        case iMovableAlert:
            doExplicitlyDeactivateDocument();
            if(!doMovableModalAlert())
            {
                SysBeep(10);
                ExitToShell();
            }
            break;

            break;

        case iModalDialog:
            doExplicitlyDeactivateDocument();
            if(!doModalDialog())
            {
                SysBeep(10);
                ExitToShell();
            }
            break;

        case iMovableModalDialog:
            doExplicitlyDeactivateDocument();
            if(!doMovableModalDialog())
            {
                SysBeep(10);
                ExitToShell();
            }
            break;

        case iModeless:
            if(!doCreateOrShowModelessDialog())
            {
                SysBeep(10);
                ExitToShell();
            }
            break;
    }
}

// ##### doExplicitlyDeactivateDocument

void doExplicitlyDeactivateDocument(void)
{
    if(FrontWindow() && ((WindowPeek) FrontWindow())->windowKind != kDialogWindowKind)
        doActivateDocument(FrontWindow(),false);
}

// ##### doMovableModalAlert

Boolean doMovableModalAlert(void)
{
    AlertStdAlertParamRec paramRec;
    Str255                labelText;
    Str255                narrativeText;
    OSErr                 osError;
    SInt16                itemHit;

    paramRec.movable      = true;

```

```

paramRec.helpButton      = false;
paramRec.filterProc     = eventFilterUPP;
paramRec.defaultText    = (StringPtr) kAlertDefaultOKText;
paramRec.cancelText     = (StringPtr) kAlertDefaultCancelText;
paramRec.otherText      = NULL;
paramRec.defaultButton  = kAlertStdAlertOKButton;
paramRec.cancelButton   = kAlertStdAlertCancelButton;
paramRec.position       = kWindowDefaultPosition;

GetIndString(labelText,rAlertStrings,sLabel);
GetIndString(narrativeText,rAlertStrings,sNarrative);

osError = StandardAlert(kAlertCautionAlert,labelText,narrativeText,&paramRec,&itemHit);
if(osError == noErr)
    return true;
else
    return false;
}

// ##### doModalDialog

Boolean doModalDialog(void)
{
    DialogPtr    modalDialogPtr;
    ControlHandle controlHdl;
    SInt16      itemHit, controlValue;

    if(!(modalDialogPtr = GetNewDialog(rModal,NULL,(WindowPtr) -1)))
        return(false);

    SetDialogDefaultItem(modalDialogPtr,kStdOkItemIndex);
    SetDialogCancelItem(modalDialogPtr,kStdCancelItemIndex);

    GetDialogItemAsControl(modalDialogPtr,iGridSnap,&controlHdl);
    SetControlValue(controlHdl,gGridSnap);
    GetDialogItemAsControl(modalDialogPtr,iShowGrid,&controlHdl);
    SetControlValue(controlHdl,gShowGrid);
    GetDialogItemAsControl(modalDialogPtr,iShowRulers,&controlHdl);
    SetControlValue(controlHdl,gShowRule);

    ShowWindow(modalDialogPtr);

    do
    {
        ModalDialog(eventFilterUPP,&itemHit);

        if(itemHit == iGridSnap || itemHit == iShowGrid || itemHit == iShowRulers)
        {
            GetDialogItemAsControl(modalDialogPtr,itemHit,&controlHdl);
            SetControlValue(controlHdl,!GetControlValue(controlHdl));
        }
        else if(itemHit == iFont || itemHit == iSound)
        {
            GetDialogItemAsControl(modalDialogPtr,itemHit,&controlHdl);
            controlValue = GetControlValue(controlHdl);
            doPopupMenuChoice(controlHdl,controlValue,itemHit);
        }
    } while((itemHit != kStdOkItemIndex) && (itemHit != kStdCancelItemIndex));

    if(itemHit == kStdOkItemIndex)
    {
        GetDialogItemAsControl(modalDialogPtr,iGridSnap,&controlHdl);
        gGridSnap = GetControlValue(controlHdl);
        GetDialogItemAsControl(modalDialogPtr,iShowGrid,&controlHdl);
        gShowGrid = GetControlValue(controlHdl);
        GetDialogItemAsControl(modalDialogPtr,iShowRulers,&controlHdl);
        gShowRule = GetControlValue(controlHdl);
    }

    DisposeDialog(modalDialogPtr);
    doCopyPString("\pBalloon help is available for dialog boxes",gCurrentString);

    return(true);
}

// ##### doMovableModalDialog

```

```

Boolean doMovableModalDialog(void)
{
    DialogPtr      movableModalDialogPtr;
    ControlHandle controlHdl;
    SInt16         oldBrushType, itemHit, a;

    if(!(movableModalDialogPtr = GetNewDialog(rMovable,NULL,(WindowPtr) -1)))
        return(false);

    SetDialogDefaultItem(movableModalDialogPtr,kStdOkItemIndex);
    SetDialogCancelItem(movableModalDialogPtr,kStdCancelItemIndex);
    SetDialogTracksCursor(movableModalDialogPtr,true);

    GetDialogItemAsControl(movableModalDialogPtr,gBrushType,&controlHdl);
    SetControlValue(controlHdl,kControlRadioButtonCheckedValue);

    GetDialogItemAsControl(movableModalDialogPtr,iClockOne,&controlHdl);
    SetKeyboardFocus(movableModalDialogPtr,controlHdl,kControlClockPart);

    oldBrushType = gBrushType;

    ShowWindow(movableModalDialogPtr);

    do
    {
        ModalDialog(eventFilterUPP,&itemHit);

        if(itemHit >= iCharcoal && itemHit <= iChalk)
        {
            for(a=iCharcoal;a<=iChalk;a++)
            {
                GetDialogItemAsControl(movableModalDialogPtr,a,&controlHdl);
                SetControlValue(controlHdl,kControlRadioButtonUncheckedValue);
            }

            GetDialogItemAsControl(movableModalDialogPtr,itemHit,&controlHdl);
            SetControlValue(controlHdl,kControlRadioButtonCheckedValue);
            gBrushType = itemHit;
        }
    } while((itemHit != kStdOkItemIndex) && (itemHit != kStdCancelItemIndex));

    if(itemHit == kStdCancelItemIndex)
        gBrushType = oldBrushType;

    DisposeDialog(movableModalDialogPtr);
    return(true);
}
// ##### doCreateOrShowModelessDialog

Boolean doCreateOrShowModelessDialog(void)
{
    Boolean      booleanData;
    ControlHandle controlHdl;
    Str255      stringData = "\pwicked googly";

    if(gSearchModelessDialogPtr == NULL)
    {
        if(!(gSearchModelessDialogPtr = GetNewDialog(rModeless,gPreAllocatedBlockPtr,
            (WindowPtr) -1)))
            return(false);

        SetWRefCon(gSearchModelessDialogPtr,(SInt32) kSearchModeless);

        booleanData = true;
        GetDialogItemAsControl(gSearchModelessDialogPtr,kStdOkItemIndex,&controlHdl);
        SetControlData(controlHdl,kControlNoPart,kControlPushButtonDefaultTag,
            sizeof(booleanData),(Ptr) &booleanData);

        GetDialogItemAsControl(gSearchModelessDialogPtr,iEditText,&controlHdl);
        SetDialogItemText((Handle) controlHdl,stringData);
        SelectDialogItemText(gSearchModelessDialogPtr,iEditText,0,32767);

        ShowWindow(gSearchModelessDialogPtr);
    }
    else

```

```

    {
        ShowWindow(gSearchModelessDialogPtr);
        SelectWindow(gSearchModelessDialogPtr);
    }

    return(true);
}

// ##### doInContent

void doInContent(EventRecord *eventStrucPtr)
{
    WindowPtr windowPtr;
    SInt16    whichModeless;
    DialogPtr dialogPtr;
    SInt16    itemHit;

    windowPtr = FrontWindow();

    if(!(IsDialogEvent(eventStrucPtr)))
    {
        // Handle clicks in document window content region here.
    }
    else
    {
        if((whichModeless = ((WindowPeek) windowPtr)->refCon) == kSearchModeless)
        {
            if(DialogSelect(eventStrucPtr,&dialogPtr,&itemHit))
                if(itemHit == kStdOkItemIndex)
                    doButtonHitInSearchModeless();
        }
    }
}

// ##### doButtonHitInSearchModeless

void doButtonHitInSearchModeless(void)
{
    ControlHandle controlHdl;
    WindowPtr    oldPort;

    GetDialogItemAsControl(gSearchModelessDialogPtr,iEditText,&controlHdl);
    GetDialogItemText((Handle) controlHdl,gCurrentString);

    GetPort(&oldPort);
    SetPort(gWindowPtr);
    doDrawMessage(gWindowPtr,false);
    SetPort(oldPort);
}

// ##### doHideModelessDialog

void doHideModelessDialog(WindowPtr windowPtr)
{
    SInt16    whichModeless;

    HideWindow(windowPtr);

    if((whichModeless = ((WindowPeek) windowPtr)->refCon) == kSearchModeless)
        gSleepTime = MAXLONG;
}

// ##### eventFilter

pascal Boolean eventFilter(DialogPtr dialogPtr,EventRecord *eventStrucPtr,SInt16 *itemHit)
{
    Boolean handledEvent;
    GrafPtr oldPort;

    handledEvent = false;

    if((eventStrucPtr->what == updateEvt) &&
        ((WindowPtr) eventStrucPtr->message != dialogPtr))
    {
        doUpdate(eventStrucPtr);
    }
}

```

```

else
{
    GetPort(&oldPort);
    SetPort(dialogPtr);

    handledEvent = StdFilterProc(dialogPtr,eventStrucPtr,itemHit);

    SetPort(oldPort);
}
return(handledEvent);
}

// ##### doPopupMenuChoice

void doPopupMenuChoice(ControlHandle controlHdl,SInt16 controlValue,SInt16 itemHit)
{
    MenuHandle menuHdl;
    Size actualSize;
    Str255 itemName;
    GrafPtr oldPort;

    GetControlData(controlHdl,kControlNoPart,kControlPopupMenuHandleTag,
        sizeof(menuHdl),(Ptr) &menuHdl,&actualSize);
    GetMenuItemText(menuHdl,controlValue,itemName);
    doCopyPString(itemName,gCurrentString);

    GetPort(&oldPort);
    SetPort(gWindowPtr);
    doDrawMessage(gWindowPtr,false);
    SetPort(oldPort);

    if(itemHit == iSound)
        doPlaySound(itemName);
}

// ##### doPlaySound

void doPlaySound(Str255 sndResourceName)
{
    SndListHandle soundHdl;
    SndChannelPtr soundChanPtr = NULL;

    if(soundHdl = (SndListHandle) GetNamedResource('snd ',sndResourceName))
        SndPlay(soundChanPtr,soundHdl,1);
}

// ##### doDrawMessage

void doDrawMessage(WindowPtr windowPtr,Boolean inState)
{
    Rect headerRect;
    SInt16 windowHeight, stringWidth;

    SetRect(&headerRect,windowPtr->portRect.left - 1,windowPtr->portRect.top - 1,
        windowPtr->portRect.right + 1,windowPtr->portRect.top + 26);
    DrawThemeWindowHeader(&headerRect,inState);

    if(inState == kThemeStateActive)
        SetThemeTextColor(kThemeTextColorWindowHeaderActive,gPixelDepth,gIsColourDevice);
    else
        SetThemeTextColor(kThemeTextColorWindowHeaderInactive,gPixelDepth,gIsColourDevice);

    windowHeight = (windowPtr->portRect.right - (windowPtr->portRect.left);
    stringWidth = StringWidth(gCurrentString);
    MoveTo((windowWidth / 2) - (stringWidth / 2), 17);
    DrawString(gCurrentString);
}

// ##### doCopyPString

void doCopyPString(Str255 sourceString,Str255 destinationString)
{
    SInt16 stringLength;

    stringLength = sourceString[0];
}

```

```

BlockMove(sourceString + 1,destinationString + 1,stringLength);
destinationString[0] = stringLength;
}

// ##### doGetDepthAndDevice

void doGetDepthAndDevice(void)
{
    GDHandle deviceHdl;

    deviceHdl = LMGetMainDevice();
    gPixelFormat = (*(deviceHdl->gdPMap)->pixelSize);
    if(BitTst(&(deviceHdl->gdFlags,gdDevType))
        gIsColourDevice = true;
}

// #####

```

Demonstration Program Comments

When this program is run, the user should:

- Invoke alert and dialog boxes by choosing items in the Demonstration menu, noting window update/activation/deactivation and menu enabling/disabling effects.
- Choose Show Balloons from the Help menu and pass the cursor over the various items in the dialog boxes, noting the information in the help balloons. Also note the updating of alert and dialog boxes, and of the window, behind the help balloon when the balloon closes.
- Note the effects on the Apple, Help, and Application menus when the various alert and dialog boxes are the front window.
- Click anywhere outside the modal 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 alert box and movable modal dialog box are displayed:
 - The program can be sent to the background by clicking outside the alert or dialog box and the document window or by selecting another application from the Application menu.
 - The program can be brought to the foreground again by clicking inside the alert or dialog box, or the document window, or by selecting the program from the Application menu.
- Note that, when the movable modal dialog box is displayed, the Edit menu and its Cut, Copy, and Paste items are displayed whenever the edit text field has keyboard focus.
- 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 program 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.
 - A modal alert box, movable modal alert box, modal dialog box or movable modal dialog box can be invoked "on top of" the modeless dialog box.
 - The Edit menu and its Cut, Copy, Paste, and Clear items are enabled so as to support text editing in the edit text field.
- Note that all alert and dialog boxes respond correctly to Return and Enter key presses, and that the modal alert box, modal dialog box and movable modal dialog box also respond correctly to escape key and Command-period presses.

- Note that, when an alert box or dialog box is the frontmost window, the window and content region are deactivated, the latter evidenced by dimming of the text in the document window's window header and the drawing of the header in the deactivated mode.
- In the modal dialog box, click on the checkboxes to change their settings, noting that the new settings are remembered when the dialog box is dismissed using the OK button, but not remembered when the dialog box is dismissed using the Cancel button. Also, choose items in the two pop-up menu buttons, noting that the chosen item is displayed in the document window's window header.
- In the movable modal dialog box, click on the radio buttons to change their settings, noting that the new setting is remembered when the dialog box is dismissed using the OK button, but not remembered when the dialog box is dismissed using the Cancel button. In the case of the clock control and edit text field, change the item/part with keyboard focus using the Tab key or by clicking in that item/part. In the case of the edit text field, enter text, and edit that text using the Edit menu's Cut, Copy, Paste, and Clear items and their Command-key equivalents. Note that the cursor shape changes whenever the cursor is moved over the edit text field.
- In the modeless dialog box, enter text, and edit that text using the Edit menu's Cut, Copy, Paste, and Clear items and their Command-key equivalents. Note that, because no cursor adjustment function is included in the program, the cursor shape does not change whenever the cursor is moved over the edit text field. Also note that, when the Search button is clicked (or the Return or Enter keys are pressed) the text in the edit text field is displayed in the document window's window header.

In the 'DITL' resources for the modal and movable modal dialog boxes, note that the item numbers of the primary group box items are lower than the item numbers of the items visually contained by those group box items. This is to ensure that the group boxes do not draw over, and thus erase, the image of these contained items.

In the 'alrx' resources, note that all feature flags are set except for the `kAlertFlagsAlertIsMovable` flag in the 'alrx' resource for the modal alert box. In the 'dlgx' resources, note that all feature flags are set except for the `kDialogFlagsHandlesMovableModal` flag in 'dlgx' resources for the modal dialog box and modeless dialog box. Thus all alert and dialog boxes have a root control and embedding hierarchy, and are fully Appearance-compliant.

Although this program only opens one modeless dialog box, a unique value is assigned to the associated window structure's `refCon` field to illustrate the usual method for differentiating between the several modeless dialog boxes a program could open at any one time.

#define

The first #defines establish constants for menu bar, window, and menu resource IDs, and for menu IDs and menu item numbers. Constants are then established for alert and dialog resource IDs and for the item numbers of certain items in the item lists associated with the three dialogs. `rAlertStrings` represents the resource ID of a 'STR#' holding strings for the label and narrative text for the movable alert box, and the two following constants are used to index these strings. The value represented by `kSearchModeless` will be assigned to the modeless dialog box's window structure's `refCon` field.

The penultimate block of #defines establish constants representing the character codes for the Return, Enter, escape, and period keys.

Finally, `MAXLONG` is defined as the maximum possible long value. This value will be assigned to `WaitNextEvent`'s `sleep` parameter at program launch.

Global Variables

`gPreAllocatedBlockPtr` will be assigned a pointer to a pre-allocated block of memory for the modeless dialog box's dialog structure.

`eventFilterUPP` will be assigned a universal procedure pointer to an application-defined event filter function.

`gCurrentString` will be assigned advisory and other strings for display in the document window's window header. `gWindowPtr` will be assigned the pointer to the single window opened by the program. `gPixelDepth` will be assigned the pixel depth of the graphics port. `gIsColourDevice` will be assigned true if the graphics device is a colour device and false if it is a monochrome device. The values in these last two variables are required by the Appearance Manager function `SetThemeTextColor`.

`gSleepTime` will be assigned the value to 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 next three variables will contain the current setting of the checkboxes in the modal dialog box. The next variable will contain the identity of the newly selected radio button in the movable modal dialog box.

Finally, the pointer to the dialog structure for the modeless dialog box is declared as a global variable.

main

After the call to `doInitManagers`, and provided Mac OS 8.5 or later is present, `GetNewDialog` is called to create a small modal dialog. `SetDialogTimeout` is then called with 10 (seconds) passed in the `inSecondsToWait` parameter and 1 passed in the `inButtonToPress` parameter. (In the associated 'DITL' resource, Item 1 is the OK push button, which has been hidden.) The use of `SetDialogTimeout` requires that the application handle events for the dialog box through the `ModalDialog` function, hence the `ModalDialog` do-while loop. This allows the Dialog Manager to simulate an item selection. After 10 seconds, the Dialog Manager simulates a user click in the (invisible) OK button, causing the do-while loop to exit. The dialog is then disposed of.

The modeless dialog will be created when the user chooses the Modeless Dialog item in the Demonstration menu, and will remain in existence until the program terminates. To avoid heap fragmentation effects, the nonrelocatable block for the modeless dialog's dialog structure is pre-allocated here, before the system software managers are initialised, so as to ensure that it is located as low in the heap as possible.

After the system software managers are initialised, a call to `NewModalFilterProc` creates a routine descriptor for the event filter function. (If this program was required to be compiled as 68K code only, this routine descriptor would not be required.)

`RegisterAppearanceClient` is called to ensure that the new Appearance-compliant menu bar definition function (resource ID 63) will be used regardless of whether system-wide Appearance is selected on or off in the Appearance control panel.

The next block sets up the drop-down menus.

The call to the application-defined function `doCopyPString` causes the string in the first parameter to be copied to the global variable `gCurrentString`. The string in `gCurrentString`, which will be changed at various points in the code, will be drawn in the document window's window header.

The next block opens a window and sets the font size for the window to 10pt.

The call to the application-defined function `doGetDepthAndDevice` determines the current pixel depth of the graphics port, and whether the current graphics device is a colour device, and assigns the results to the global variables `gPixelDepth` and `gIsColourDevice`.

The main event loop is then entered, and continues until `gDone` is set to true.

Note that error handling here and in other areas of this demonstration program is somewhat rudimentary. In the unlikely event that certain calls fail, `ExitToShell` is called to terminate the program.

eventLoop

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

The variable which will be used as `WaitNextEvent`'s sleep parameter (`gSleepTime`) is initially set to `MAXLONG`, 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 at certain points in the program.

doIdle

`doIdle` is invoked whenever `WaitNextEvent` returns a null event.

If the front window is the modeless dialog box, the function `IdleControls` is called. `IdleControls` calls the control definition function of those controls in the specified window which do idle-time processing. In this case, the control is an edit text field, and the call causes the control definition function to call `TextEdit` to blink the insertion point caret.

doEvents

doEvents switches according to the event type received. (It is important to remember at this point that events that occur when an alert box, modal dialog box, or movable modal dialog box has been invoked are not handled by the main event loop but by the ModalDialog function.)

doMouseDown

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

If the mouse-down occurred in the content region, and if it was not in the frontmost window, SelectWindow is called 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.

If a mouse-down occurred in a close box, if TrackGoAway returns true, and if the window is a modeless dialog box, the application-defined function doHideModeless is called. (In this demonstration, the modeless dialog box, but not the document window, has a close box.) Also, the text for the window header is replaced with the default advisory text.

doKeyDown

doKeyDown handles all key-down and auto-key events, switching according to whether the event occurred in the modeless dialog box or the document window.

First, the character code is extracted from the message field of the event structure. Then IsDialogEvent is called to determine whether the event occurred in a modeless dialog box or a document window.

If the event occurred in a document window, and if the modifiers field of the event structure indicates that the Command key was down, the application-defined function for adjusting the menus is called, MenuEvent is called to return the long value containing the menu and menu item associated with the Command-key equivalent, and the long value is passed to doMenuChoice for further handling.

If, however, the event occurred in a modeless dialog box, and if that dialog box is the Search modeless dialog box:

- If the key pressed was the Return or Enter key, GetDialogItemAsControl is called to get a handle to the single press button control in the Search modeless dialog box (item 1 in the item list). The press button is then highlighted for eight ticks, and then unhighlighted before an application-defined function is called to extract the text from the edit text field and display it in the document window's window header. doKeyDown then returns because it is not intended that the edit text field receive Return and Enter key presses.
- If the Command key was down:
 - If either the X, C, or V key was pressed (that is, the user has pressed the Cut, Copy, or Paste Command-key equivalent), DialogSelect is called to further handle the event. DialogSelect uses TextEdit to cut, copy, or paste the text in the edit text field. (The calls to HiliteMenu briefly highlight the Edit menu to indicate to the user that an Edit menu Command-key equivalent has just been used. This replicates the highlighting that ModalDialog performs when Command-key presses occur in modal and movable modal dialog boxes with edit text fields.)
 - If neither the X, C, nor V key was pressed, the application-defined function for adjusting the menus is called, MenuEvent is called to return the long value containing the menu and menu item associated with the Command-key equivalent, and the long value is passed to doMenuChoice for further handling.

Thus the Command-key equivalents other than those for Cut, Copy, and Paste remain available to the user via the main event loop, while the Command-key equivalents for Cut, Copy, and Paste are trapped and passed to DialogSelect for handling. At the last line in the outer if block, doKeyDown returns if the Command key was down.

- If the Return key and the Enter key were not pressed, and if the Command key was not down, DialogSelect is called to handle the keystroke in conjunction with TextEdit, the visual result being that the character appears in the edit text field.

doUpdate

doUpdate performs the initial handling of update events.

If the call to `IsDialogEvent` reveals that the event is for a window of the document kind, an application-defined function for updating the document window is called, otherwise an application-defined function for updating modeless dialog boxes is called.

doUpdateDocument

`doUpdateDocument` simply fills the content region of the document window with a colour, using a pixel pattern ('ppat') resource and a call to `FillRect` for that purpose, and then calls an application-defined function which draws a window header frame, and some text, in the appropriate mode (activated or deactivated) depending on whether the window is the frontmost window or not.

doUpdateModelessDialog

`doUpdateModelessDialog` calls `DialogSelect` to handle the update event. `DialogSelect` calls `BeginUpdate`, `DrawDialog`, and `EndUpdate` to redraw the modeless dialog's content area. (To restrict the redraw to the update region, an alternative is to call `BeginUpdate`, `UpdateDialog`, and `EndUpdate`.)

doActivate

`doActivate` performs initial handling of activate events.

If the call to `IsDialogEvent` reveals that the event is for a window of the document kind, the application-defined function for activating/deactivating the document window is called, otherwise the application-defined function for activating/deactivating a modeless dialog box is called.

doOSEvent

`doOSEvent` handles operating system events, .

If the event is a suspend/resume event, `doOSEvent` calls the appropriate window activation function (depending on whether the event is for the document window or a modeless dialog), indicating with the value in `gInBackground` whether the window should be activated or deactivated.

doActivateDocument

`doActivateDocument` performs window activation/deactivation for the document window.

If the window is becoming active, the menus are adjusted as appropriate for a document window. The call to the application-defined function `doDrawMessage` draws a window header frame in the window, and some advisory text, in either the activated or deactivated mode.

doActivateModelessDialog

`doActivateModelessDialog` performs window activation and deactivation for a modeless dialog box.

`DialogSelect` is called to handle the event. If the modeless dialog box is becoming active, `DialogSelect` activates all controls and redraws the one-pixel-wide modeless dialog frame in the undimmed mode. If the modeless dialog box is going to the back, `DialogSelect` deactivates all controls and redraws the one-pixel-wide modeless dialog frame in the dimmed mode.

At the next block, if the modeless dialog box is becoming active, and if it is the Search modeless dialog box (identified by a unique value assigned by the application to the dialog window structure's `refCon` field), the global variable used in the sleep parameter of the `WaitNextEvent` function is assigned 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). This is necessary to ensure that null events will always be generated, and thus `doIdle` and `IdleControls` will be called, at an interval short enough to ensure insertion point caret blinking at the proper rate. (The differentiation between modeless dialog boxes in this instance is for illustrative purposes only, since the program only ever opens one modeless dialog box. Other modeless dialog boxes might not necessarily contain edit text fields.)

If the Search modeless dialog box is to be deactivated, the sleep parameter for the `WaitNextEvent` function is reset to the maximum long value.

doAdjustMenus

doAdjustMenus is called by the document window and modeless dialog box activation functions to adjust the menus as appropriate to the type of the frontmost window, that is, whether the frontmost window is the document window or the modeless dialog box.

doMenuChoice

doMenuChoice handles menu choices.

If the choice was the Quit item in the File menu, gDone is set to true, thus terminating the program. If the choice was the Close item in the File menu, and if the front window is a modeless dialog box, an application-defined function which hides modeless dialog boxes is called. (In this program, and because the document window does not have a close box, the Close item is only enabled when the modeless dialog box is the front window.)

doEditMenu

doEditMenu first determines whether the front window is a modeless dialog box, and whether it is the Search modeless dialog (which has an edit text field). If so, Cut, Copy, Paste, and Clear selections from the Edit menu will cause the Dialog Manager, in conjunction with TextEdit, to perform those operations on selected text in the edit text field.

doDemonstrationMenu

doDemonstrationMenu handles choices from the Demonstration menu, switching according to the menu item passed to it. (Error handling in this function is somewhat rudimentary in that the program simply terminates.)

If the user chose Modal Alert, StopAlert is called to create, display, manage, and dispose of the modal alert box. Before invoking any type of alert box, however, an application must explicitly deactivate the front document window, if one exists. Accordingly, an application-defined function is called to perform that action. Note that a universal procedure pointer to an application-defined event filter function is passed in StopAlert's second parameter. StopAlert handles all user interaction within the modal alert box, disposing of the alert box when the user clicks the OK button or presses the Return key.

If the user chose Movable Modal Alert, the application-defined function which deactivates the frontmost document window (if one exists) is called, following which the application defined function which creates, displays, handles user interaction, and disposes of the movable modal alert box is called. (As will be seen, for the movable modal alert box also uses the application-defined event filter function.)

if the user chose Modal Dialog or Movable Modal Dialog, the same general procedure is followed, except that the application-defined function which creates, displays, handles user interaction, and disposes of the modal dialog or movable modal dialog is called. (As will be seen, for the modal dialog box and movable modal dialog box also use the application-defined event filter function.)

If the user chose Modeless Dialog, the application-defined function for creating and displaying the modeless dialog box is called.

doExplicitlyDeactivateDocument

doExplicitlyDeactivateDocument is called prior to the opening of all but the modeless dialog box to explicitly deactivate the frontmost document window (if one exists).

If there is at least one window of any type open, and if that front window is of the document window kind, the application-defined function for activating/deactivating document windows is called to deactivate the window.

doMovableModalAlert

doMovableModalAlert creates, displays, manages, and disposes of the movable modal alert box. Unlike the modal alert box, and all the dialog boxes, the movable modal dialog box in this demonstration is created programmatically, rather than from 'ALRT' and 'alrx' resources.

At the first nine lines, values are assigned to the fields of a standard alert structure. In sequence: the alert box is to be a movable modal alert box; a help button is not to be displayed; the event filter function used is to be the application-defined event filter function pointed to by the universal procedure pointer eventFilterUPP; the default text for the OK button is to be used; the default text for the Cancel button is to be used; no "left-most" button is required; the default push button is to be the first push button (which will thus have the default ring drawn around it and have the Return and Enter keys aliased to it); the

Cancel push button is to be the second push button (which will thus have escape and Command-period key presses aliased to it); the alert box is to be displayed in the alert position on the parent window screen. (With regard to the last field, in this structure, the constant `kWindowDefaultPosition` equates to `kWindowAlertPositionParentWindowScreen`.)

The two calls to `GetIndString` retrieve the specified strings from the specified 'STR#' resource. These are passed in the label text and narrative text parameters in the following call to `StandardAlert`.

The call to `StandardAlert` creates and displays the alert, handles all user interaction (by internally calling `ModalDialog`), including dismissing the alert box when either the OK or Cancel buttons are hit. (The item hit is returned in `StandardAlert`'s last parameter. In a real application, the appropriate action would be taken, based on which item was hit, following the call to `StandardAlert`.)

doModalDialog

`doModalDialog` creates, displays, manages, and disposes of the modal dialog box.

The call to `GetNewDialog` creates the modal dialog box from the specified resource as the frontmost window.

The call to `SetDialogDefaultItem` tells the Dialog Manager which is the default push button item, to alias the Return and Enter keys to that item, and to draw the default ring around that item. The call to `SetDialogCancelItem` tells the Dialog Manager which is the Cancel push button item, and aliases the escape key and Command-period key presses to that item.

The next block gets handles to the three checkbox controls and sets the value of those controls to the current values contained in the global variables relating to each control.

With the modal dialog box fully prepared, it is made visible by the call to `ShowWindow`.

The do/while loop continues to execute until `ModalDialog` reports that either the OK or Cancel button has been "hit". Within the loop, `ModalDialog` retains control until one of the enabled items has been hit.

If a checkbox is hit, `GetDialogItemAsControl` is called to get a handle to the control and `SetControlValue` is called to flip that control's control value. (If it is 0, it is flipped to 1, and vice versa.) If one of the pop-up menu buttons is hit, `GetDialogItemAsControl` is called to get a handle to the control and `GetControlValue` is called to get the menu item number of the menu item chosen, following which an application defined function is called to extract the menu item text and display it in the window header. (In the case of the Sound pop-up menu button, an additional function is called to play the associated 'snd ' resource.)

Note that the first parameter in the `ModalDialog` call is a universal procedure pointer to the application-defined event filter function.

When the do/while loop exits, and if the user hit the OK button, 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 created and displayed.)

The dialog is then disposed of, and true is returned to the calling function. (The call to `doCopyPString` simply causes the text displayed in the window header text to be replaced with the default advisory text.)

doMovableModalDialog

`doMovableModalDialog` creates, displays, manages, and disposes of the movable modal dialog box.

The call to `GetNewDialog` creates the movable modal dialog box from the specified resource as the frontmost window.

The call to `SetDialogDefaultItem` tells the Dialog Manager which is the default push button item, to alias the Return and Enter keys to that item, and to draw the default ring around that item. The call to `SetDialogCancelItem` tells the Dialog Manager which is the Cancel push button item, and to alias the escape key and Command-period key presses to that item. The call to `SetDialogTracksCursor` tells the Dialog Manager to track the cursor and change it to the I-Beam cursor shape whenever it is over an edit text field.

The first call to `GetDialogItemAsControl` gets a handle to the radio button control represented by the current value in the global variable `gBrushType`. The value of that control is then set to 1. The second call to `GetDialogItemAsControl` gets a handle to the clock control. The call to `SetKeyboardFocus` sets the keyboard focus to that item.

Before the session of user interaction begins, the current value in the global variable `gBrushType`, which stores the item number of the currently selected radio button, is copied to the local variable `oldBrushType`.

With the movable modal dialog box fully prepared, it is made visible by the call to `ShowWindow`.

The `do/while` loop continues to execute until `ModalDialog` reports that either the OK or Cancel button has been "hit". Within the loop, `ModalDialog` retains control until one of the enabled items is hit.

If a radio button is hit, a `for` loop sets the control value of all radio button controls to 0. A call to `GetDialogItemAsControl` then gets a handle to the radio button control that was hit. A call to `SetControlValue` then sets that control's value to 1, and the item number of this radio button is assigned to the global variable `gBrushType`.

Note that the first parameter in the `ModalDialog` call is a universal procedure pointer to the application-defined event filter function. Note also that all user interaction relating to the clock control and edit text field is handled automatically by `ModalDialog`, including the movement of keyboard focus between the items.

When the `do/while` loop exits, and if the user hit the Cancel button, the value stored in the local variable `oldBrushType` is assigned to `gBrushType`, ensuring that any change to the currently selected radio button within the `do/while` loop is ignored. (In a real application, a long date/time value from the clock control, and the text from the edit text field would possibly be retrieved at this point if the user hit the OK push button.)

The dialog is then disposed of, and `true` is returned to the calling function.

doCreateOrShowModelessDialog

In this program, the modeless dialog box is only created once, that is, when the user first chooses Modeless Dialog from the Demonstration menu. Clicks in its close box, or choosing 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, the first line determines whether the modeless dialog box is already open. If it is not: the call to `GetNewDialog` creates the modeless dialog box (with the previously pre-allocated nonrelocatable block passed in the second parameter); the call to `SetWRefCon` assigns the constant `kSearchModeless` to the `refCon` field of the dialog's window structure so as to facilitate the future identification of this particular modeless dialog box; a call to `SetControlData` with the `kControlPushButtonDefaultTag` tag constant causes the control definition function to draw the default ring around the specified push button; a call to `SetDialogItemText` assigns some initial text to the edit text field; a call to `SelectDialogItemText` selects the text in the edit text field. (Note that, if the edit text field did not contain text, this latter call would simply display the insertion point caret, which would be made to blink by the call to `IdleControls` within the application-defined function `doIdle`.)

If, on the other hand, the modeless dialog box has already been opened, the call to `ShowWindow` displays the dialog box and the call to `SelectWindow` generates the necessary activate events.

doInContent

`doInContent` continues the content region mouse-down handling initiated by `doMouseDown`. `doInContent` is called by `doMouseDown` only if the mouse-down occurred in the frontmost (active) window.

The first line gets a pointer to the frontmost window.

If the event occurred in the document window, the mouse-down event would be handled in the `if/else` block. (No action is required in this demonstration.)

If the event occurred in a modeless dialog box, and if that modeless dialog box is the Search modeless dialog box (which contains an edit text field), `DialogSelect` is called to handle the event. `DialogSelect` tracks enabled controls (only the push button is enabled), returning `true` if the mouse button is released while the cursor is still inside the control, and highlights any selection made in the edit text field. If `DialogSelect` returns `true`, and if the item hit was the OK (Search) push button, an application-defined function is called to perform the actions required in the event of a hit on that button.

doButtonHitInSearchModeless

`doButtonHitInSearchModeless` further processes, to completion, a hit on the OK (Search) button in the modeless dialog box. It simply demonstrates retrieval of the text in an edit text field in a dialog box.

The call to `GetDialogItemAsControl` gets a handle to the edit text field, and the call to `GetDialogItemText` copies the text in the edit text field to the global variable `gCurrentString`. The following lines cause that text to be drawn, in the dimmed mode, in the window header in the document window.

doHideModelessDialog

`doHideModelessDialog` hides a modeless dialog box.

The call to `HideWindow` makes the dialog box invisible. If the modeless dialog box is the Search modal dialog box, the sleep parameter for the `WaitNextEvent` function is reset to the maximum possible long value, because insertion point caret blinking is not required while the Search dialog box is hidden.

eventFilter

`eventFilter` is the application-defined event filter function which, in conjunction with `ModalDialog`, handles events in all alert and dialog boxes except the modeless dialog box. In this program, a universal procedure pointer to `eventFilter` is passed as the first parameter in the `StopAlert` and `ModalDialog` calls.

If the event is an update event, and if it is not for the dialog box or alert box in question, the application's document window updating function is called, and false is returned. This response to an update event in the application's own document windows also allows `ModalDialog` to perform a minor switch when necessary so that background applications can update their windows as well.

If the event is not an update event, the current graphics port is saved and then set to that of the alert or dialog box. The event is then passed to the standard event filter function for handling. If the standard event filter function handles the event, it will return true and, in the `itemHit` parameter, the number of the item that it handled. `ModalDialog` (and `StopAlert`, which calls `Modal Dialog` internally) will then return this item number. A call to `SetPort` then restores the previously save graphics port.

Note that the calls to `GetPort` and `SetPort` are actually redundant when this event filter function is used by all but the movable modal dialog box. The calls are only necessary when `SetDialogTracksCursor` has been called to cause the Dialog Manager to automatically track the cursor, and the movable modal dialog box is the only modal alert or dialog box which requires this tracking (because it contains an edit text field.)

doPopupMenuChoice, doPlaySound, doDrawMessage, doCopyPString, and doGetDepthAndDevice

`doPopupMenuChoice`, `doPlaySound`, `doDrawMessage`, `doCopyPString`, and `doGetDepthAndDevice` are incidental to the demonstration. All perform the same duties as the similarly-named functions in the demonstration program `Controls1`, which is associated with Chapter 7 – Introduction to Controls. `doDrawMessage` is used in this program to prove the explicit deactivation of the document window's content area when alert and dialog boxes other than the modeless dialog box are invoked.

EDIT TEXT CONTROL BACKGROUND COLOUR PROBLEM

There was an apparent bug in the version of the edit text control CDEF included with Versions 1.0 through 1.0.3 of the Appearance Manager, but which is not evident in the version of the CDEF included with Mac OS 8.5. This bug manifested itself in the edit text item in the modeless dialog of the demonstration program as corruption of the text background colour whenever a cut, clear, or paste was made via the Edit menu (though not via the Command-key equivalents).

If you are running this demonstration under a version of the Mac OS earlier than 8.5, this bug will almost certainly be evident. One workaround is to force a re-draw of the edit text item in the modeless dialog immediately after `DialogCut`, `DialogClear`, and `DialogDelete` are called. Proceed as follows:

Add this function prototype:

```
void doFixEditTextBackground(void);
```

Add this function:

```
// ##### doFixEditTextBackground
```

```
void doFixEditTextBackground(void)
{
    ControlHandle controlHdl;

    GetDialogItemAsControl(gSearchModelessDialogPtr,iEditText,&controlHdl);
    DeactivateControl(controlHdl);

    ActivateControl(controlHdl);
}
```

In the doEditMenu function, add a call to doFixEditTextBackground immediately after the calls to DialogCut, DialogClear, and DialogDelete.