8

DIALOGS AND ALERTS

Demonstration Program: DialogsAndAlerts

Introduction

Your application should present alerts to the user whenever an unexpected or undesirable situation occurs.

Before your application carries out a command, it may present a dialog to solicit additional information from the user or to allow the user to modify settings.

Alert Types, Modalities, and Levels

Alert Types and Modalities

There are three types of alert, namely, the **modal alert**, the **movable modal alert**, and, on Mac OS X only, the **sheet alert**. The three types are shown at Fig 1.

Modal Alert

The fixed-position modal alert places the user in the state, or **mode**, of being able to work only inside the alert. The only response the user receives when clicking anywhere outside the alert is the alert sound. The modal alert is thus **system-modal**, meaning that is denies user interaction with anything but the alert until it is dismissed.

There will be very few, if any, situations where the use of a modal alert in your application is justified.

Movable Modal Alert

Movable modal alerts retain the essentially modal characteristic of their fixed-position counterpart, the main differences being that they allow the user to drag the alert so as to uncover obscured areas of an underlying window and bring another application to the front. Movable modal alerts are thus **application-modal**.

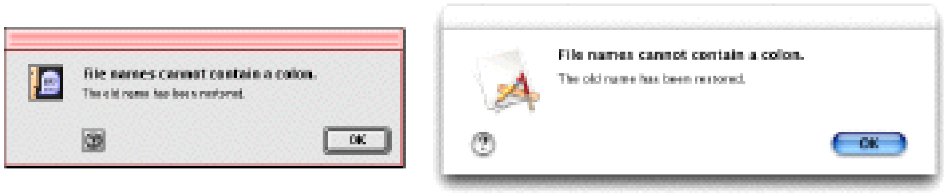
Window-Modal (Sheet) Alert — Mac OS X

Mac OS X introduced a new type of alert called the **sheet alert**. Sheet alerts, which are invariably attached to an owner window, are **window-modal**. The information conveyed by the alert, or the alternative actions requested, should pertain only to the document to whose window the alert is attached.

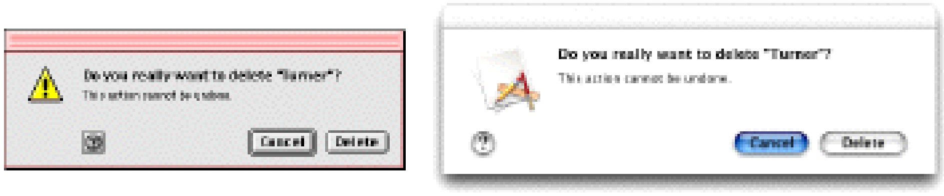
Levels of Alert

Modal and movable modal alerts can display one of three levels of alert (see Fig 1), depending on the nature of the situation the alert is reporting to the user. The three levels of alert, which, on Mac OS 8/9, are identified by icons supplied automatically by the system, are as follows:

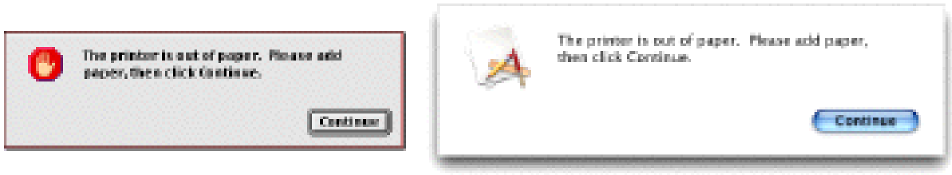
- ***Note Level.*** The note level (see Fig 1) is used to inform users of an occurrence that will not have serious consequences. Usually, a note level alert simply offers information, although it may ask a simple question and provide, via the push buttons, a choice of responses.
- ***Caution Level.*** The caution level is used to alert the user to an operation that 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 with, or back out of, the operation.
- ***Stop Level.*** The stop level is used to inform the user of a situation so serious that the operation cannot proceed.


File names cannot contain a colon.
The old name has been restored.

MOVABLE MODAL ALERT (NOTE LEVEL)

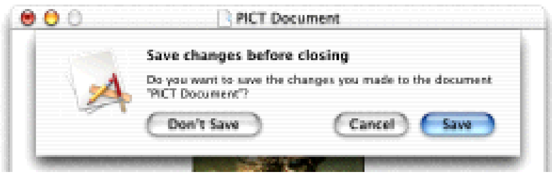

Do you really want to delete "Turner"?
This action cannot be undone.

MOVABLE MODAL ALERT (CAUTION LEVEL)


The printer is out of paper. Please add paper, then click Continue.

MODAL ALERT (STOP LEVEL)

The Mac OS X window layering model, in which document windows from different applications can be interleaved, makes it necessary to indicate to the user which application is displaying the alert. This is achieved by the inclusion of the application's icon in the alert.


Save changes before closing
Do you want to save the changes you made to the document "PICT Document"?

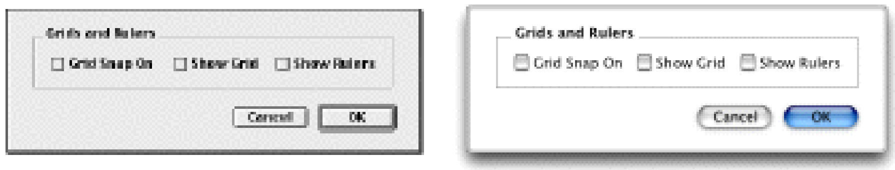
WINDOW-MODAL (SHEET) ALERT

FIG 1 - TYPES AND LEVELS OF ALERTS

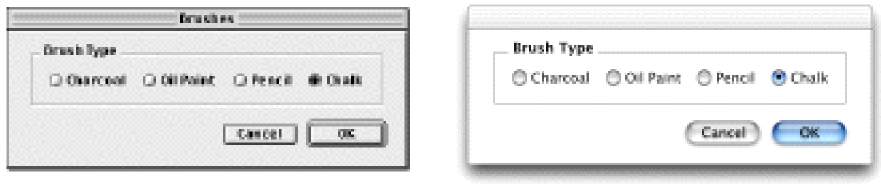
Note that, at the time of writing, there was no visual distinction between alert levels on Mac OS X, the application icon rather than distinct note, caution, and stop icons being displayed. At the time of writing, it was expected that Carbon would eventually support the "badging" of the application icon with alert level badges similar to the Mac OS 8/9 note, caution, and stop icons.

Dialog Types and Modalities

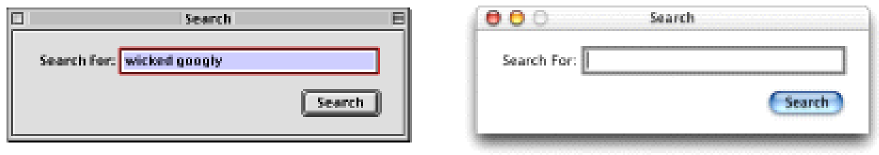
There are four types of dialog, namely, modal dialogs, movable modal dialogs, modeless dialogs, and, on Mac OS X only, sheet dialogs. The four types are illustrated in the examples at Fig 2.



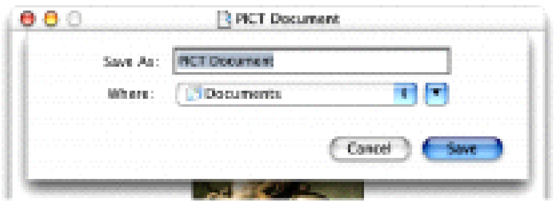
MODAL DIALOG



MOVABLE MODAL DIALOG



MODELESS DIALOG



WINDOW-MODAL (SHEET) DIALOG

FIG 2 - TYPES OF DIALOGS

Modal Dialog

Fixed-position modal dialogs place the user in the state, or mode, of being able to work only inside the dialog. The only response the user receives when clicking outside the dialog is the alert sound. The modal alert is thus **system-modal**, meaning that is denies user interaction with anything but the dialog until it is dismissed.

There will be very few, if any, situations where the use of a modal dialog in your application is justified.

Movable Modal Dialog

Movable modal dialogs retain the essentially modal characteristic of their fixed-position counterpart, the main differences being that they allow the user to drag the dialog so as to uncover obscured areas of an underlying window and bring another application to the front. Movable modal dialogs are thus **application-modal**.

The absence of boxes/buttons in the title bar of a movable modal dialog visually indicates to the user that the dialog is modal rather than modeless.

Modeless Dialog

Modeless dialogs look very like document windows, except for their interior colour/pattern and, on Mac OS 8/9, a one-pixel frame just inside the window frame. Unlike document windows, however, modeless dialogs should not contain scroll bars or a size box/resize control.

Modeless dialogs should not require the user to dismiss them before the user is able to do anything else. Thus modeless dialogs should be made to behave much like document windows in that the user should be able to move them, bring other windows in front of them, and close them.

Modeless dialogs should ordinarily not have a **Cancel** push button, although they may have a **Stop** push button to halt long operations such as searching.

Window-Modal (Sheet) Dialog — Mac OS X

Mac OS X introduced a new type of dialog called the **sheet dialog**. Sheet dialogs, which are invariably attached to an owner window, are **window-modal**. The information or settings solicited by the dialog should pertain only to the document to whose window the dialog is attached.

Window Types For Alerts and Dialogs

Fig 3 shows the seven (eight on Mac OS X) available 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.

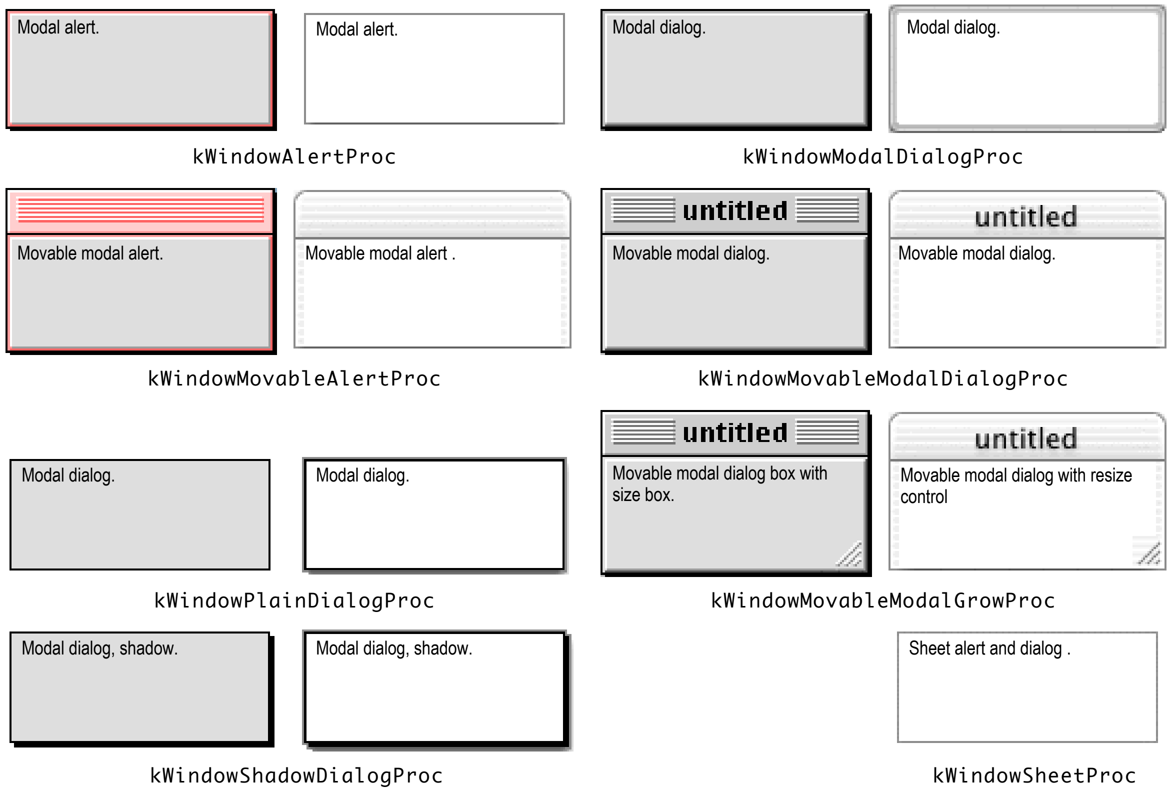


FIG 3 - WINDOW TYPES FOR DIALOGS AND ALERTS

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:

WDEF Resource ID	*Variation Code*	*Window Definition ID (Value)*	*Window Definition ID (Constant)*
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` (Used for modeless dialogs.)
68	0	`68 * 16 + 0 = 1088`	`kWindowSheetProc`

Content of Alerts and Dialogs

Alerts should usually contain only informative text and push button controls. Dialogs may contain informative or instructional text and controls.

Default Push Buttons

Your application should specify a **default push button** for every alert and dialog. The default push button, visually identified by a default ring drawn around it (Mac OS 8/9) or pulsing blue (Mac OS X), should be the one the user is more likely to click in most circumstances. 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. (See the caution alert at Fig 1.)

Removing Dialogs

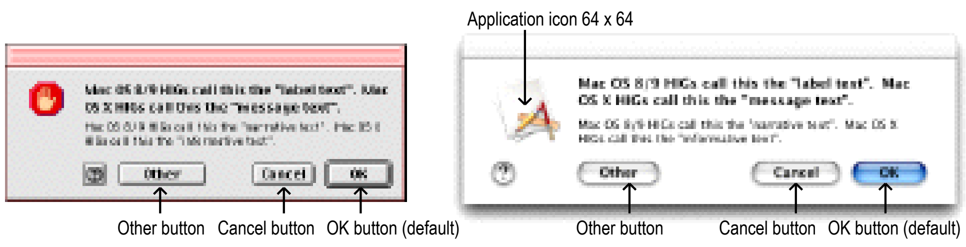
Your application should remove and dispose of a modal, movable modal, and window-modal (sheet) dialogs only when the user clicks one of its push buttons.

Your application should not remove a modeless dialog unless the user clicks its close box/button or chooses **Close** from the **File** menu when the modeless dialog is the active window. (Typically, a modeless dialog is simply hidden, not disposed of, when the user clicks the close box/button or chooses **Close** from the **File** menu.)

Creating and Removing Alerts

Alerts may be created from resources using the functions `Alert`, `NoteAlert`, `CautionAlert` and `StopAlert`, which take descriptive information about the alert from **alert** (`'ALRT'`) and **extended alert** (`'alrx'`) **resources**. However, the preferred (and considerably simpler) method is to create alerts programmatically using the functions `StandardAlert` and, on Mac OS X only, `CreateStandardAlert`.

Fig 4 shows an alert created by `StandardAlert` and `CreateStandardAlert`.


Mac OS 8/9 HIGs call this the "label text". Mac OS X HIGs call this the "message text".

Mac OS 8/9 HIGs call this the "narrative text". Mac OS X HIGs call this the "informative text".

Mac OS 8/9 HIGs call this the "label text". Mac OS X HIGs call this the "message text".

Mac OS 8/9 HIGs call this the "narrative text". Mac OS X HIGs call this the "informative text".

FIG 4 - ALERTS CREATED WITH StandardAlert AND CreateStandardAlert

The StandardAlert Function

When an alert is created using the function `StandardAlert`, the alert is automatically sized based on the amount of text passed in to it, and push buttons are automatically sized and located.

```
OSErr  StandardAlert(AlertType inAlertType,ConstStr255Param inError,
                     ConstStr255Param inExplanation,
                     const AlertStdAlertParamRec *inAlertParam,SInt16 *outItemHit);
```

`inAlertType`	The level of alert. Relevant constants are: `kAlertStopAlert` `kAlertNoteAlert` `kAlertCautionAlert` `kAlertPlainAlert`
`inError`	The label text (Mac OS 8/9) or message text (Mac OS X).
`inExplanation`	The narrative text (Mac OS 8/9) or informative text (Mac OS X). `NULL` indicates no narrative/informative text.
`inAlertParam`	A pointer to a **standard alert parameter structure** (see below). `NULL` indicates that none of the features provided by the standard alert structure are required.
`outItemHit`	On return, contains the item number of the push button that the user hit.

Standard Alert Parameter Structure

The standard alert parameter structure is as follows:

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

Field Descriptions

`movable`	Specifies whether the alert is modal or movable modal.
`helpButton`	Specifies whether the alert should include the Help button.
`filterProc`	Optionally, a universal procedure pointer to an application-defined event filter (callback) function. If `NULL` is assigned, the Dialog Manager uses the standard event filter (callback) function. (See Event Filter (Callback) Functions For Modal and Movable Modal Alerts and Dialogs, below).
`defaultText`	Optionally, text for the push button in the OK push button[1], position. (See Alert Default Text Constants, below). The push button is automatically sized and positioned to accommodate the text.
`cancelText`	Optionally, text for the push button in the **Cancel** push button position. (See Alert Default Text Constants, below.) The push button is automatically sized and positioned to accommodate the text. Pass `NULL` to specify that a Cancel push button should not be displayed.

[1] The push button in the **OK** button position is not necessarily named **OK**. Human Interface Guidelines require that, wherever possible, buttons be named with a verb that describes the action that they perform. (As an example, see the buttons at Fig 1.)

`otherText`	Optionally, text for the push button in leftmost position. (See Alert Default Text Constants, below.) The push button is automatically sized and positioned to accommodate the text. Pass `NULL` to specify that the leftmost push button should not be displayed.
`defaultButton`	Specifies which push button is to act as the default push button. (See Alert Push Button Constants, below.)
`cancelButton`	Specifies which push button is to act as the **Cancel** push button. Can be `0`. (See Alert Button Constants, below.)
`position`	The alert position. (See Positioning Specification, below, and note that, when these constants are used to specify alert position in an alert created programmatically using `StandardAlert`, the constant `kWindowDefaultPosition` has the same effect as `kWindowAlertPositionParentWindowScreen`.)

Alert Default Text Constants

To specify the default text for the push buttons in the Right, Middle, and Leftmost push button positions, use these constants in the `defaultText`, `cancelText`, and `otherText` fields of the standard alert structure:

Constant	*Value*	*Button Position*	*Default Text*	*Where Used*
`kAlertDefaultOKText`	-1	Right	**OK**	`defaultText` field
`kAlertDefaultCancelText`	-1	Middle	**Cancel**	`cancelText` field
`kAlertDefaultOtherText`	-1	Leftmost	**Don't Save**	`otherText` field

Alert Push Button Constants

To specify which push buttons act as the default and Cancel push buttons, use these constants in the `defaultButton` and `cancelButton` fields in the standard alert structure:

Constant	*Value*	*Meaning*
`kAlertStdAlertOKButton`	1	The OK push button.
`kAlertStdAlertCancelButton`	2	The Cancel push button.
`kAlertStdAlertOtherButton`	3	A third push button.

Positioning Specification

The main constants for the positioning specification field are as follows:

Constant	*Value*	*Menaing*
`kWindowDefaultPosition`	0x0000	Alert position on screen where user is currently working.
`kWindowAlertPositionMainScreen`	0x300A	Alert position on main screen. (The main screen in a multi-monitor system is the screen on which the menu bar is located.)
`kWindowAlertPositionParentWindow`	0xB00A	Alert position on frontmost window.
`kWindowAlertPositionParentWindowScreen`	0x700A	Alert position on screen where user is currently working.
`kWindowCenterMainScreen`	0x280A	Centre on main screen.
`kWindowCenterParentWindow`	0xA80A	Centre on frontmost window.
`kWindowCenterParentWindowScreen`	0x680A	Centre on screen where user is currently working.

The CreateStandardAlert Function

On Mac OS X only, you may also use the function `CreateStandardAlert` to create an alert:

```
OSStatus  CreateStandardAlert(AlertType inAlertType, CFStringRef inError,
                              CFStringRef inExplanation,
                              const AlertStdCFStringAlertParamRec *param,
                              DialogRef *outAlert);
```

The main differences between the `CreateStandardAlert` and `StandardAlert` functions are as follows:

- The `inError` and `inExplanation` fields take a `CFStringRef`.
- A pointer to a **standard CFString alert parameter structure** is passed in the `param` parameter. This structure is basically similar to the **standard alert parameter structure** except that:
    - It has no field for a universal procedure pointer to an event filter (callback) function.
    - The `defaultText`, `cancelText`, and `otherText` fields are of type `CFStringRef`.
    - It has an additional field (`flags`) in which bits can be set to specify options for the behaviour of the dialog. Setting the `kStdAlertDoNotDisposeSheet` bit in this field when the dialog is a sheet causes the sheet not to be disposed of after it is hidden.

  A call to the function `GetStandardAlertDefaultParams` initialises a standard CFString alert parameter structure with default values. (The defaults are: not movable; no Help button; no Cancel button; no Other button; alert position on parent window screen.)
- On return, a pointer to a dialog reference is received in the `outAlert` parameter. This must be passed in a call to the function `RunStandardAlert`, which displays the alert and handles user interaction. A universal procedure pointer to an event filter (callback) function may be passed in the `filterProc` parameter of this function. On return, the item number of the push button that the user hit is received in the `outItemHit` parameter of `RunStandardAlert`.

Removal of Alerts

The Dialog Manager automatically removes and disposes of an alert when the user clicks a push button.

Creating Dialogs

Dialogs may be created in one of two ways, as follows:

- You can create dialogs from resources 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.
- You can create dialogs programmatically using the function `NewFeaturesDialog`. `NewFeaturesDialog` has a `flags` parameter containing the same flags you would set in an extended dialog resource when creating the dialog from resources.

Regardless of which method is used to create the dialog, a **dialog object** will be created, and a pointer to that object will be returned to the calling function. The dialog object itself includes a window object.

The Dialog Object

Dialog objects are opaque data structures in which the Dialog Manager stores information about individual dialogs. The data type `DialogRef` is defined as a pointer to a dialog object:

```
typedef struct OpaqueDialogPtr *DialogPtr;
typedef DialogPtr DialogRef;
```

Accessor Functions

The following accessor functions are provided to access the information in dialog objects.

Function	Description
`GetDialogWindow`	Gets a reference to the dialog's window object.
`GetDialogTextEditHandle`	Gets a handle to the `TERec` structure (which is re-used for all edit text items).
`GetDialogKeyboardFocusItem`	Gets item number of the item with keyboard focus.
`GetDialogDefaultItem`	Gets the item number of the default push button.
`SetDialogDefaultItem`	Tells the Dialog Manager the item number of the default push button.
`GetDialogCancelItem`	Gets the item number of the default Cancel push button.
`SetDialogCancelItem`	Tells the Dialog Manager the item number of the default Cancel button.

`AppendDITL`
`AppendDialogItemList`
`ShortenDITL`
`InsertDialogItem`
`RemoveDialogItem`
	Add items to, and remove items from, a dialog.

'DLOG' and 'dlgx' Resources

Structure of a Compiled 'DLOG' Resource

Fig 5 shows the structure of a compiled `'DLOG'` resource and how it "feeds" the dialog object.

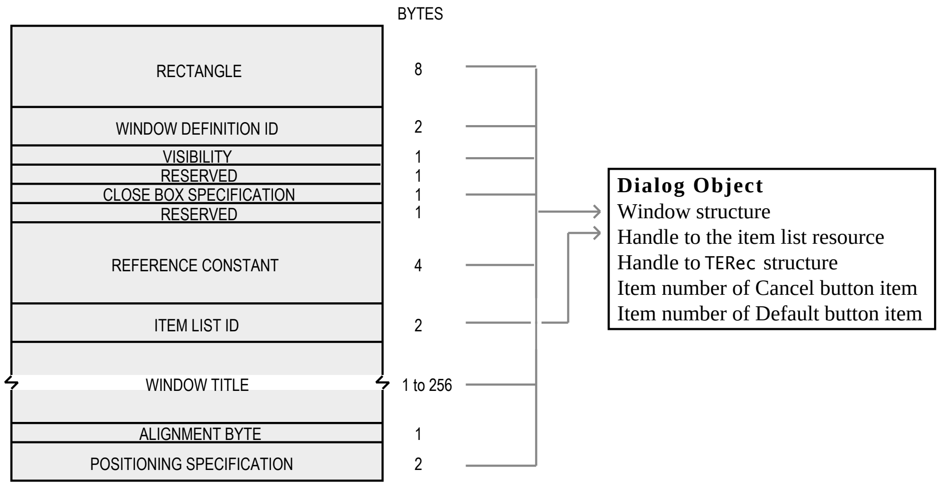


FIG 5 - STRUCTURE OF A COMPILED DIALOG ('DLOG') RESOURCE

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

Field	Description
RECTANGLE	The dialog's dimensions and, if a positioning specification (see below) is not specified, its location.
WINDOW DEFINITION ID	The window definition ID. (See Window Types For Alerts and Dialogs, above.)
VISIBILITY	If set to 1, the Dialog Manager displays the dialog as soon as `GetNewDialog` is called. If set to 0, the dialog is not displayed until `ShowWindow` is called.
CLOSE BOX SPECIFICATION	Specifies whether to draw a close box/button. Ordinarily, a close box/button is specified only for modeless dialogs.
REFERENCE CONSTANT	Applications can store any value here. For example, an application might store a number that represents the dialog type. `SetWRefCon` and `GetWRefCon` tmay be used to set and get this value.
ITEM LIST ID	Resource ID of the item list resource.
WINDOW TITLE	The title displayed in the dialog's title bar (modeless and movable modal dialogs only).

POSITIONING SPECIFICATION	Specifies the position of the dialog on the screen. If a positioning constant is not provided, the Dialog Manager places the dialog at the global coordinates specified for the dialog's rectangle (see above). The same positioning constants as apply in the case of an alert apply. (See Positioning Specification, above, but note that, in the case of `'DLOG'` resources, `kWindowDefaultPosition` means that the window will be positioned according to the RECTANGLE field.)

Structure of a Compiled 'dlgx' Resource

Fig 6 shows the structure of a compiled `'dlgx'` resource. This resource allows you to provide additional features for your dialog, including movable modal behaviour, background colour/pattern, and embedding hierarchies.

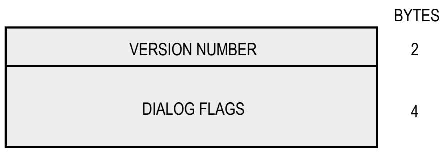


FIG 6 - STRUCTURE OF A COMPILED DIALOG ('dlgx') RESOURCE

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

Field	*Description*
DIALOG FLAGS	Constants that specify the dialog’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's features:

Constant	*Bit*	*Meaning If Set*
`kDialogFlagsUseThemeBackground`	0	The Dialog Manager sets the correct dialog background colour/pattern.
`kDialogFlagsUseControlHierarchy`	1	A root control is created and an embedding hierarchy is established. **Note:** All items in a dialog automatically become controls when embedding hierarchy is established.
`kDialogFlagsHandleMovableModal`	2	The dialog will be movable modal (in which case you must use `kWindowMovableModalDialogProc` window definition ID). (The Dialog Manager handles movable modal behaviour.)
`kDialogFlagsUseThemeControls`	3	All controls created by the Dialog Manager will be compliant with the Platinum appearance.

Creating 'dlgx ' and 'DLOG' Resources Using Resorcerer

Creating 'dlgx' Resources

Fig 7 shows a `'dlgx'` resource being created with Resorcerer.

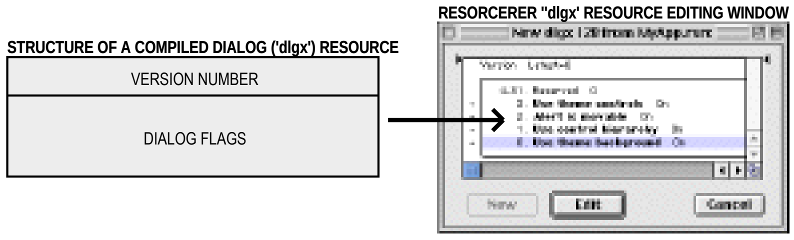
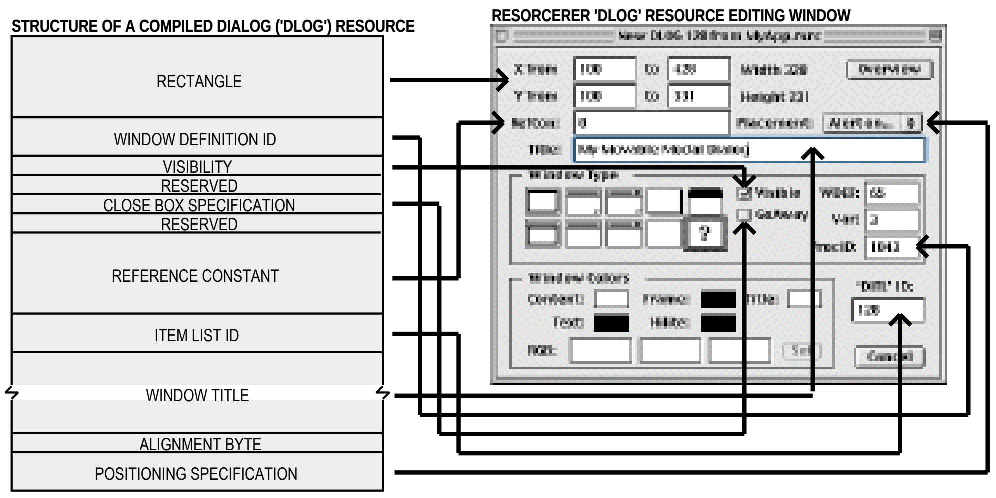

RESORCERER "dlgx' RESOURCE EDITING WINDOW
STRUCTURE OF A COMPILED DIALOG ('dlgx') RESOURCE
VERSION NUMBER
DIALOG FLAGS
New
Edit
Cancel

FIG 7 - CREATING A 'dlgx' RESOURCE USING RESORCERER

Creating 'DLOG' Resources

Fig 8 shows a `'DLOG'` resource being created with Resorcerer.



X from: 100 to: 428
Y from: 100 to: 331
RefCon: 0
Title: My Movable Modal Dialog
WDEF: 65
Vari: 2
ProcID: 1042
'DITL' ID: 128

FIG 8 - CREATING A 'DLOG' RESOURCE USING RESORCERER

The NewFeaturesDialog Function

The function `NewFeaturesDialog` creates a dialog from the information passed in its parameters.

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

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

`inStorage`	A pointer to the memory for the dialog object. In Carbon, this should always be set to `NULL`, which causes the Dialog Manager to automatically allocate memory for the dialog object.
`inBoundsRect`	A rectangle which specifies the size and position of the dialog in global coordinates.
`inTitle`	The title of a modeless or movable modal dialog. You can specify an empty string (not `NULL`) if the dialog is to have no title. (In C, you specify an empty string by two double quotation marks (`""`).)
`inIsVisible`	Specifies whether the dialog should be drawn immediately after `NewFeatureDialog` is called. If this parameter is set to `false`, `ShowWindow` must be called to display the dialog.
`inProcID`	The window definition ID for the type of dialog. Pass `kWindowModalDialogProc` in this parameter to specify modal dialogs, `kWindowMovableModalDialogProc` to specify movable modal dialogs, and `kWindowDocumentProc` to specify modeless dialogs.
`inBehind`	A reference to the window behind which the dialog is to be opened. Pass `(WindowRef) -1` in this parameter to open the dialog in front of all other windows.
`inGoAwayFlag`	Passing `true` in this parameter causes a close box/button to be drawn in the title bar. `true` should only be passed when a modeless dialog is being created.
`inRefCon`	A reference constant that the Dialog Manager stores in the dialog's window object. Applications can store any value here. For example, an application might

store a number that represents the dialog type. `GetWRefCon` may be used to retrieve this value.

`inItemListHandle`	A handle to the item list resource, which you can get by calling `GetResource` to read the item list resource into memory.
`inFlags`	The dialog's feature flags. (See Dialog Feature Flag Constants, above.)

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

Items in Dialogs

Preamble — Dialog Manager Primitives

Dialogs 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**.

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

The situation where all items in a dialog are controls has many advantages. For example, all controls within the dialog can be activated and deactivated by simply activating and deactivating the root control.

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

Dialog Manager Primitive	*Control Equivalent*
Button.	Push button.
Radio Button.	Radio button.
Checkbox.	Checkbox.
Edit Text.	Edit text control.
Static Text.	Static text control.
Icon. (An icon whose black and white resource is stored in an `'ICON'` resource and whose colour version is stored in a `'cicn'` resource with the same ID.)	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.)	(No control equivalent.)

The 'DITL' Resource

A (`'DITL'`) resource is used to store information about all the items in a dialog. The `'DITL'` resource ID is specified in the associated `'DLOG'` resource, or a handle to the `'DITL'` resource is passed in the `inItemListHandle` parameter of the `NewFeaturesDialog` function. `'DITL'` resources should be marked as purgeable.

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

Several independent dialogs may use the same `'DITL'` resource. `AppendDITL`, `AppendDialogItemList`, and `ShortenDITL` may be used to modify or customise copies of shared item list resources for use in individual dialogs.

Fig 9 shows the structure of a compiled `'DITL'` resource and one of its constituent items, in this case a control item.

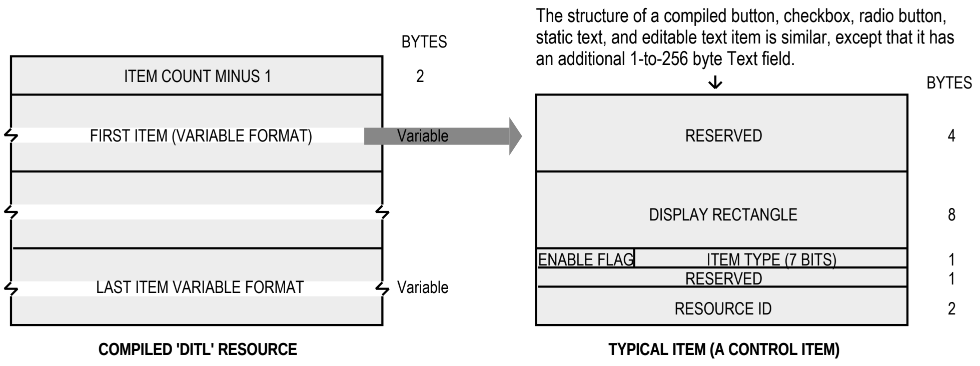

The structure of a compiled button, checkbox, radio button, static text, and editable text item is similar, except that it has an additional 1-to-256 byte Text field.
BYTES
ITEM COUNT MINUS 1
2
FIRST ITEM (VARIABLE FORMAT)
Variable
LAST ITEM VARIABLE FORMAT
Variable
BYTES
RESERVED
4
DISPLAY RECTANGLE
8
ENABLE FLAG
ITEM TYPE (7 BITS)
1
RESERVED
1
RESOURCE ID
2
COMPILED 'DITL' RESOURCE
TYPICAL ITEM (A CONTROL ITEM)

FIG 9 - 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:

Field	*Description*
ITEM COUNT MINUS 1	A value equal to one less than the number of items in the resource.
FIRST ITEM ... LAST ITEM	(The format of each item depends on its type.)
DISPLAY RECTANGLE	The size and location, in local coordinates, of the item within the dialog. (See Display Rectangles, below.)
ENABLE FLAG	Specifies whether the item is enabled or disabled. If this bit is set (item is enabled) the Dialog Manager reports all mouse-down events in the item to your application
ITEM TYPE	The item type.
RESOURCE ID	For a control item, the resource ID of the `'CNTL'` resource.

Display Rectangles

The enclosing rectangle you specify in a control's `'CNTL'` resource should be identical to the display rectangle specified in the `'DITL'` resource.[2]

Note that, for items that are controls, the rectangle added to the update region is the rectangle defined in the `'CNTL'` resource, not the display rectangle specified in the `'DITL'` resource. Other important aspects of display rectangles are as follows:

- ***Edit Text Items.*** In edit text items, the display rectangle is the TextEdit **destination rectangle** and **view rectangle** (see Chapter 21). For display rectangles that are large enough to contain more than one line of text, word wrapping occurs within the rectangle. The text is clipped if it overflows the rectangle.
- ***Static Text Item.*** Static text items are drawn within the display rectangle in the same manner as edit text items except that a frame is not drawn around the text.
- ***Icon and Picture Items.*** Icons and pictures larger than the display rectangle are scaled so as to fit the display rectangle.
- The Dialog Manager considers a click anywhere in the display rectangle to be a click in that item. In the case of a click in the overlap area of overlapping display rectangles, the Dialog Manager reports the click as occurring in the item that appears first in the item list.

Creating a 'DITL' Resource Using Resorcerer

Fig 10 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.

[2] 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.

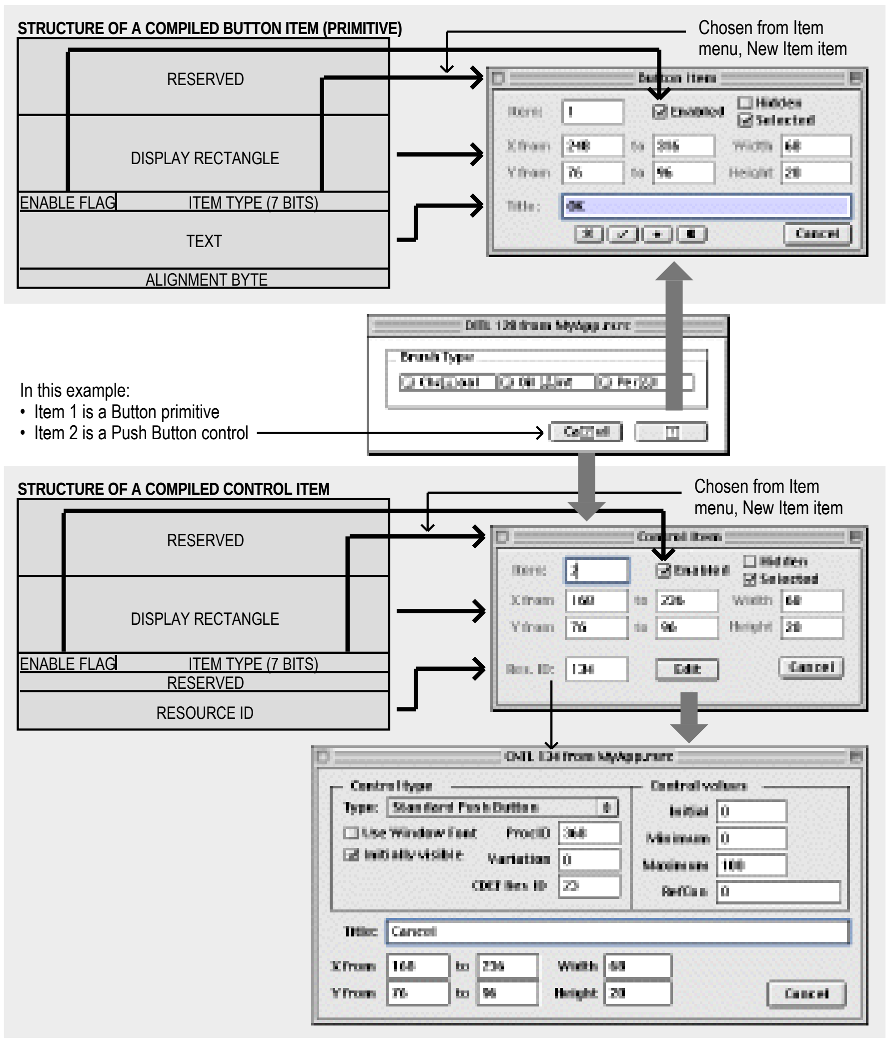

STRUCTURE OF A COMPILED BUTTON ITEM (PRIMITIVE)
RESERVED
DISPLAY RECTANGLE
ENABLE FLAG
ITEM TYPE (7 BITS)
TEXT
ALIGNMENT BYTE
Chosen from Item menu, New Item item
In this example:
• Item 1 is a Button primitive
• Item 2 is a Push Button control
STRUCTURE OF A COMPILED CONTROL ITEM
RESERVED
DISPLAY RECTANGLE
ENABLE FLAG
ITEM TYPE (7 BITS)
RESERVED
RESOURCE ID
Chosen from Item menu, New Item item

FIG 10 - CREATING A 'DITL' RESOURCE USING RESORCERER

Layout Guidelines For Dialogs

Layout guidelines for items in dialogs are contained in the Apple publications Mac OS 8 Human Interface Guidelines (Mac OS 8/9) and Aqua Human Interface Guidelines (Mac OS X). These guidelines are not consistent on matters such as the required sizes of certain items and, more particularly, the required spacing between items. For Carbon applications, it is best to observe the Aqua interface guidelines when laying out dialog items.

Default Push Buttons

You should give every dialog a default push button, except for those that contain edit text items that accept Return key presses. If you do not provide an event filter (callback) function (see Event Filter (Callback) Functions For Modal and Movable Modal Alerts and Dialogs, 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.

Enabling and Disabling Items

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

Note that *disabled* is not the same thing as a *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. On the other hand, when a control is deactivated, the Control Manager dims it to show that it is deactivated.

Keyboard Focus

Edit text and clock items accept input from the keyboard, and list box items 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 items, the Dialog Manager also automatically displays the insertion point caret in the current target. For clock items, 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:

Function	*Description*
`GetDialogItemAsControl`	Returns the control reference 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 reference, 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 reference to the control. When called on a static text item, `GetDialogItem` returns a handle to the text, not a reference to the control, and thus may be used to get a handle to the text of static text items. (When called on a static text item, `GetDialogItemAsControl` returns a reference to the control, not a handle to the text.)
`SetDialogItem`	When an embedding hierarchy does not exist, sets the item type, reference, and display rectangle of an item. (When an embedding hierarchy exists, you cannot change the type or reference of an item.)
`HideDialogItem`	Hides the given item.
`ShowDialogItem`	Re-displays a hidden item.
`GetDialogItemText`	Returns the text of an edit or static text item.
`SelectDialogItemText`	Selects the text of an edit text item. When embedding is on, you should pass in the control reference produced by a call to `GetDialogItemAsControl`. When embedding is not on, you should pass in the reference produced by a call to `GetDialogItem`.
`FindDialogItem`	Determines the item number of an item at a particular location in a dialog.
`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.
`AppendDITL` `AppendDialogItemList`	Adds items to the end of the item list. (See Append Method Constants, below.)
`ShortenDITL`	Removes items from the end of the item list.
`InsertDialogItem`	Inserts an item into the item list.
`RemoveDialogItem`	Removes an item from the item list.
`ParamText`	Substitutes up to four different text strings in static text control items.

Append Method Constants

The AppendDITL, AppendDialogItemList, and ShortenDITL functions are particularly useful in the situation where more than one dialog shares the same 'DITL' resource and you want to tailor the 'DITL' for each dialog. When calling AppendDITL or AppendDialogItemList, you specify a new 'DITL' resource to append to the relevant dialog'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 or AppendDialogItemList call:

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

As an alternative to passing these constants, you can append items relative to an existing item by passing a negative number to AppendDITL or AppendDialogItemList. The absolute value of this number represents the item relative to which the new items are to be positioned. For example, -3 would cause the display rectangles of the appended items to be offset from the upper-left corner of item number 3 in the dialog.

To use, at a later time, the unmodified version of a dialog whose contents and (possibly) size have been modified by AppendDITL or AppendDialogItemList, you should call ReleaseResource to release the memory occupied by the appended item list.

Getting and Setting The Text in Edit Text and Static Text Items

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

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

The function ParamText may also be used to set the text string in a static text item in a dialog. 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 item in the 'DITL' resource.

Since there are four text replacement variables, ParamText can supply up to four text strings for a single dialog.

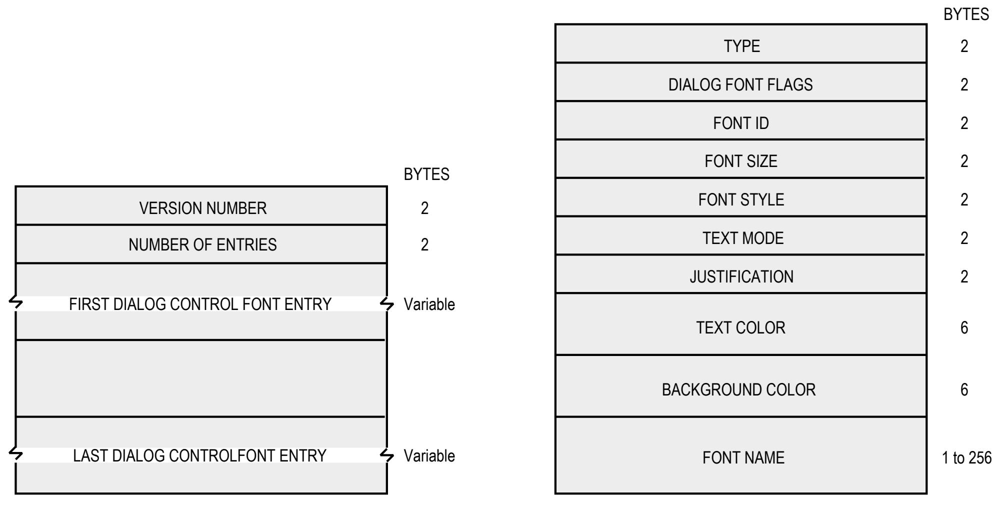
Setting the Font For Controls in a Dialog — 'dftb' Resources

When an embedding hierarchy is established in a dialog, you can specify the initial font settings for all controls in a dialog 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.

Structure of a Compiled 'dftb' Resource

Fig 11 shows the structure of a compiled 'dftb' resource and of a constituent dialog font table entry.



STRUCTURE OF A COMPILED WINDOW ('dftb') RESOURCE

DIALOG CONTROL FONT ENTRY

FIG 11- STRUCTURE OF A COMPILED DIALOG FONT TABLE ('dftb') RESOURCE AND A DIALOG CONTROL FONT ENTRY

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

Field	***Description***
NUMBER OF ENTRIES	Specifies the number of entries in the resource. Each entry is a dialog control font structure.
FIRST DIALOG CONTROL FONT ENTRY... LASTDIALOG CONTROL FONT ENTRY	Dialog control font structures. Each comprises type, dialog font flags, font ID, font size, font style, text mode, justification, text color, background color, and font name.
TYPE	Specifies whether there is font information for the dialog or alert item in the `'DITL'`. `0` means that there is no font information for the item, that no data follows, and that the entry is to be skipped. `1` means that there is font information for the item, and that the rest of the structure is read.
DIALOG FONT FLAGS	Specifies which of the following fields in the dialog font table should be used. (See Dialog Font Flag Constants, below.)
FONT ID	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`, the system default font is used.
FONT SIZE	If the `kDialogFontUseSizeMask` bit in the dialog font flags field is set, the point size of the text. If the `kDialogFontAddSizeMask` bit is set, the size to add to the current point size of the text. 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.
STYLE	The text style (normal, bold, italic,underlined, outline, shadow, condensed, or extended.)
TEXT MODE	Specifies how characters are drawn. (See Chapter 12 for a discussion of transfer modes.)
JUSTIFICATION	Justification (left, right, centered, or system-justified).
TEXT COLOR	Colour to use when drawing the text.
BACKGROUND COLOR	Colour to use when drawing the background behind the text. In certain text modes, background colour is ignored.
FONT NAME	The font name. This overrides the font ID.

Dialog Font Flag Constants

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

Constant	***Value***	***Meaning***
`kDialogFontNoFontStyle`	`0x0000`	No font style information is applied.
`kDialogFontUseFontMask`	`0x0001`	The specified font ID is applied.
`kDialogFontUseFaceMask`	`0x0002`	The specified font style is applied.
`kDialogFontUseSizeMask`	`0x0004`	The specified font size is applied.

`kDialogFontUseForeColorMask`	`0x0008`	The specified text color is applied. This flag only applies to static text controls.
`kDialogFontUseBackColorMask`	`0x0010`	The specified background color is applied. This flag only applies to static text controls.
`kDialogFontUseModeMask`	`0x0020`	The specified text mode is applied.
`kDialogFontUseJustMask`	`0x0040`	The specified text justification is applied.
`kDialogFontUseAllMask`	`0x00FF`	All flags in this mask will be set except `kDialogFontAddFontSizeMask` and `kDialogFontUseFontNameMask`.
`kDialogFontAddFontSizeMask`	`0x0100`	The specified font size will be added to the existing font size specified in the Font Size field of the dialog font table resource.
`kDialogFontUseFontNameMask`	`0x0200`	The string in the Font Name field will be used for the font name instead of the specified font ID.

Meta Font Constants

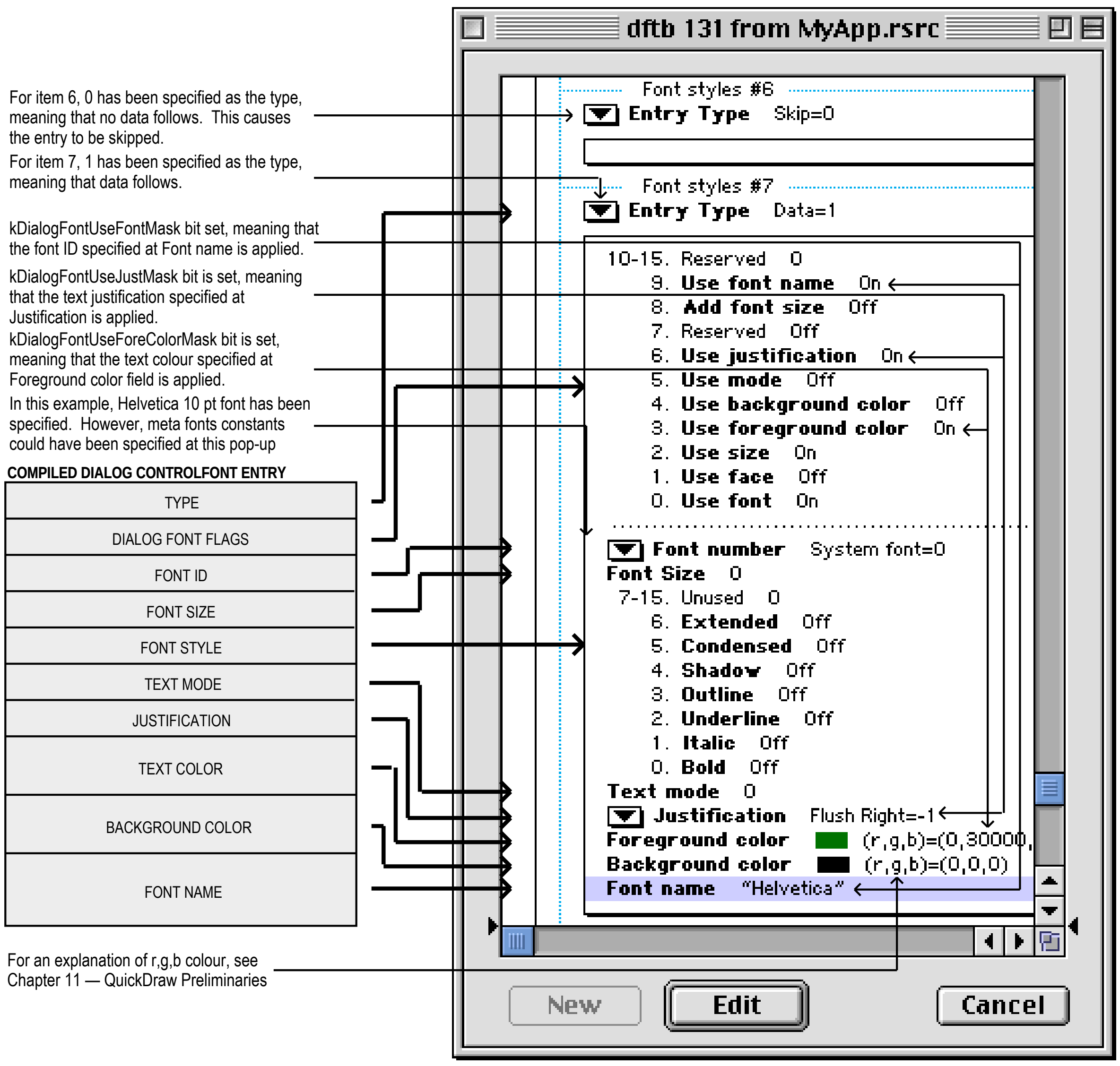
You can use the following meta font constants in the font ID field of a dialog control font entry to specify the style, size, and font family of a control's font. You should use these meta font constants whenever possible because, on Mac OS 8/9, the system font can be changed by the user. If none of these constants are specified, the control uses the system font unless a control with a variant that uses the window font has been specified.

Constant	*Value*	*Meaning In Roman Script System*
`kControlFontBigSystemFont`	`-1`	Use the system font.
`kControlFontSmallSystemFont`	`-2`	Use the small system font.
`kControlFontSmallBoldSystemFont`	`-3`	Use the small bold system font.
`kControlFontSmallBoldSystemFont`	`-4`	Use the small emphasized system font.

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.

Creating a 'dftb' Resource Using Resorcerer

Fig 12 shows a dialog control font entry in a `'dftb'` 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

For an explanation of r,g,b colour, see Chapter 11 — QuickDraw Preliminaries

```
Font styles #6
Entry Type   Skip=0

Font styles #7
Entry Type   Data=1
10-15. Reserved   0
    9. Use font name   On
    8. Add font size   Off
    7. Reserved   Off
    6. Use justification   On
    5. Use mode   Off
    4. Use background color   Off
    3. Use foreground color   On
    2. Use size   On
    1. Use face   Off
    0. Use font   On
Font number   System font=0
Font Size   0
 7-15. Unused   0
    6. Extended   Off
    5. Condensed   Off
    4. Shadow   Off
    3. Outline   Off
    2. Underline   Off
    1. Italic   Off
    0. Bold   Off
Text mode   0
Justification   Flush Right=-1
Foreground color   (r,g,b)=(0,30000,
Background color   (r,g,b)=(0,0,0)
Font name   "Helvetica"
```

FIG 12 - CREATING A 'dftb' RESOURCE USING RESORCERER

Displaying Alerts and Dialogs

As previously stated:

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

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

Window Deactivation and Menu Adjustment

When an alert or dialog 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 and dialog. (As will be seen, the system software automatically performs some of this menu adjustment for you.)

> ***Note***
>
> Prior to the introduction of Mac OS 8 and the Appearance Manager, window deactivation when a movable modal dialog was displayed was handled in the same way as applies in the case of a modeless dialog, that is, within the application's main event loop. However, with the introduction of the Appearance Manager, 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 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 adjusment 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 dialogs (as is the case with modal dialogs).
> - 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 dialogs).

Window Deactivation — Modeless Dialogs

You do not have to deactivate the front window *explicitly* when displaying a modeless dialog. 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 Alerts and Dialogs

When a modal or movable modal alert or dialog is created and displayed, your application (in the case of dialogs) or `StandardAlert` and `RunStandardAlert` (in the case of alerts) calls `ModalDialog` to handle all user interaction within the alert or dialog until the alert or 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.

Menu Adjustment — Modeless Dialogs

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

- Disable those menus whose items are not relevant to the modeless dialog.
- For modeless dialogs that contain edit text controls, enable the **Edit** menu and support the **Cut**, **Copy**, **Paste**, and **Clear** items using the Dialog Manager functions `DialogCut`, `DialogCopy`, `DialogPaste` and `DialogDelete`.

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

Menu Adjustment — Modal Alerts and Dialogs

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

- On Mac OS 8/9, the Mac OS 8/9 Application menu, and all items in the **Help** menu except the **Show Balloons**/**Hide Balloons** item are disabled. On Mac OS X, all but the Apple and Application menus are disabled.
- Your application's menus are disabled.
- If the modal dialog contains a visible and active edit text item, the **Edit** menu and its **Cut**, **Copy** and **Paste** items are enabled.

When the user dismisses the modal alert or dialog, the Menu Manager restores all menus to their previous state.

Menu Adjustment — Movable Modal Alerts and Dialogs

When your application displays a movable modal alert or dialog, 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 alerts and dialogs except that, in this case, on Mac OS 8/9, the **Help** and Mac OS 8/9 Application menus are left enabled.

When the user dismisses the movable modal alert or dialog, the Menu Manager restores all menus to their previous state.

Displaying Multiple Alerts and Dialogs

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

Resizing a Dialog

You can use the function `AutoSizeDialog` to automatically resize static text items and their dialogs to accommodate changed static text. For each static text item found, `AutoSizeDialog` adjusts the static text control and the bottom of the dialog window. Any items below a static text control are moved down.

Handling Events in Alerts and Dialogs

Overview

Modal and Movable Modal Alerts and Dialogs

When `StandardAlert`, `CreateStandardAlert`, and `RunStandardAlert` are used to create and display alerts, the Dialog Manager handles all of the events generated by the user until the user clicks a push button. When the user clicks a push button, these functions highlight the push button briefly, close the alert 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 `filterProc` field of the standard alert structure associated with the `StandardAlert` function, the `filterProc` parameter of the `RunStandardAlert` function, and the `modalFilter` parameter of the `ModalDialog` function take a universal procedure pointer to a callback function known as an **event filter function**. The Dialog Manager provides a **standard event filter function**, which is used if `NULL` is passed in the `filterProc` or `modalFilter` parameters or assigned to the `filterProc` field; however, you should supply an application-defined event filter (callback) function for modal and movable modal alerts and dialogs so as to avoid a basic limitation of the standard event filter (callback) function. (See Event Filter (Callback) Functions For Modal and Movable Modal Alerts and Dialogs, below.)

Modeless Dialogs

For modeless dialogs, you can use the function `IsDialogEvent` to determine whether the event occurred while a modeless dialog was the frontmost window and then, optionally, use the function `DialogSelect` to handle the event if it belongs to a modeless dialog. `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 (callback) 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 to reflect to the values returned by the controls.

Controls That Accept Keyboard Input

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

- A blinking vertical bar, called the **insertion point caret**, appears when the user clicks the item.
- When the user drags over text or double-clicks a word, that text is highlighted and replaced by whatever the user types.
- Highlighting of text is extended or shortened when the user holds down the Shift key while clicking or dragging.
- Highlighted text, or the character preceding the insertion point caret, is deleted when the user presses the backspace key.
- Highlighted text, or the character following the insertion point caret, is deleted when the user presses the delete key.
- When the user presses the Tab key, the cursor and keyboard focus frame automatically advance to the next edit text control, 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 Controls

`ModalDialog` will cause the insertion point caret to blink in edit text controls in modal and movable modal dialogs. On Mac OS 8/9, for edit text controls in a modeless dialog, you should call `IdleControls` in your main event loop's idle processing function. (This is not necessary on Mac OS X because controls on Mac OS X have their own built-in timers.) `IdleControls` calls the edit text control with an idle event so that the control can call `TEIdle` to make the insertion point caret blink. You should ensure that, when caret blinking is required, the `sleep` parameter in the `WaitNextEvent` call is set to a value no greater that that returned by `GetCaretTime`.

Responding to Events in Modal and Movable Modal Alerts

`StandardAlert` and `RunStandardAlert` handle events automatically, calling `ModalDialog` internally.

If the event is a mouse-down anywhere outside the content region of a modal alert, `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 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 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 clicks an enabled control, at which time `StandardAlert` and `RunStandardAlert` remove the alert from the screen and return the item number of the selected control. Your application then should then respond appropriately.

The standard event filter (callback) function allows users to use the Return or Enter key to achieve the same effect as a click on the default push button. When you write your own event filter (callback) function, you should ensure that that function retains this behaviour. `ModalDialog` passes events inside the alert to your event filter (callback) function *before* handling the event. Your event filter (callback) function thus provides a means to:

· Handle events which `ModalDialog` does not handle.

· Override events `ModalDialog` would otherwise handle.

If your event filter (callback) function does not handle an event inside an alert in its own way, `ModalDialog` handles the event as follows:

· For activate or update events, `ModalDialog` activates or updates the alert window.

· For mouse-down events 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 is removed and the control's item number is returned.

· For a mouse-down event in a disabled item, or in no item, or if any other event occurs, nothing happens.

Responding To Events in Modal and Movable Modal Dialogs

Your application should call `ModalDialog` immediately after displaying a modal or movable modal dialog. `ModalDialog` repeatedly handles events inside the dialog until an event involving an enabled item occurs, at which time `ModalDialog` exits, returning the item number. Your application should then respond appropriately. `ModalDialog` should be continually called until the user clicks on the OK, Cancel, or Don't Save push button, at which time your application should close the dialog.

If the event is a mouse-down anywhere outside the content region of a modal dialog, `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 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 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 (callback) function does not handle the event, `ModalDialog` handles the event as follows:

· For activate or update events, `ModalDialog` activates or updates the dialog window.

· If the event is a mouse-down while the cursor is in a control that accepts keyboard input (that is, an edit text control or a clock control), `ModalDialog` responds to the mouse activity by either displaying an insertion point or by selecting text in an edit text control 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 control in the dialog, `ModalDialog` uses TextEdit to handle text entry and editing automatically. For an enabled edit text control, `ModalDialog` returns its item number after it receives either the mouse-down or key-down event. (Normally, edit text controls should be disabled.)

- For mouse-down events 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 control's item number is returned.
- 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.
- For a mouse-down event in a disabled item, or in no item, or if any other event occurs, nothing happens.

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. This allows your application to specify additional events that are not by default received by `ModalDialog`, such as operating system events. If you us this function to change the `ModalDialog` function's event mask, you must pass `ModalDialog` a universal procedure pointer to your own event filter (callback) function to handle the added events.

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

Simulating Item Selection

You can cause the Dialog Manager to simulate item selection in a modal or movable modal dialog using the function `SetDialogTimeout`. You can use this function in circumstances where you wish to start a countdown for a specified duration for a specified dialog. When the specified time elapses, the Dialog Manager simulates a click on the specified button. 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`.

Event Filter (Callback) Functions For Modal and Movable Modal Alerts and Dialogs

The standard event filter (callback) function dates from the early days of the Macintosh, when a single application controlled the computer. With the introduction of multitasking, however, the standard event filter proved to be somewhat inadequate, its main deficiency being that it does not cater for the updating of either the parent application's windows or those belonging to background applications. (This deficiency is only relevant on Mac OS 8/9.) Your application should therefore provide an application-defined event filter (callback) function which compensates for this inadequacy and handles other events you wish the function to handle.

The standard event filter (callback) 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 (Mac OS 8/9 only) and returns the item number of that push button. (Unless informed otherwise, the Dialog Manager assumes that the first item in the item list is the default push button.)
- For dialogs 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 (Mac OS 8/9 only) and returns the item number of that button.
    - Check whether the cursor is over an edit text item and, if so, changes the cursor shape to the I-Beam cursor.

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

- For Mac OS 8/9 only, handle update events not belonging to the alert or dialog 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 (callback) function does not handle.

Defining an Event Filter (Callback) Function

Part of the recommended approach to defining a basic event filter (callback) function is to continue to use the standard event filter (callback) function to perform its checks and actions as described above. This requires certain preliminary action which, for dialogs, 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 (Mac OS 8/9) or
// make it pulsing blue (Mac OS X).

SetDialogDefaultItem(myDialogRef,iOK);

// 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(myDialogRef,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 item.

SetDialogTracksCursor(myDialogRef,true);
```

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

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

```
Boolean  myEventFilterFunction(DialogRef dialogRef,EventRecord *eventStrucPtr,
                               SInt16 *itemHit)
{
  Boolean handledEvent;
  GrafPtr oldPort;

  handledEvent = false;

  if((eventStrucPtr->what == updateEvt) &&
     ((WindowRef) eventStrucPtr->message != dialogRef))
  {
    // 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.

    if(!gRunningOnX)
      doUpdate(eventStrucPtr);
  }
  else
  {
    // If the event was not an update, first save the current graphics port and set the
    // alert or dialog'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);
    SetPortDialogPort(dialogRef);
```

```
      // 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, StandardAlert, and
      // RunStandardAlert then return this item number in their own itemHit parameter.

      handledEvent = StdFilterProc(dialogRef,eventStrucPtr,itemHit);

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

      SetPort(oldPort);
    }

    // Return true or false, as appropriate.

    return(handledEvent);
  }
```

`StandardAlert`, `RunStandardAlert`, and `ModalDialog` pass events to your event filter (callback) function *before* handling each event[3], and will handle the event if your event filter (callback) function returns `false`.

You can also use your event filter (callback) function to handle events that `ModalDialog` does not handle, such as keyboard equivalents and mouse-down events.

Responding to Events in Modeless Dialogs

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

· For activate or update events, `DialogSelect` activates or updates the modeless dialog and returns `false`.

· If the event is a key-down or auto-key event, and there is an edit text item in the modeless dialog, `DialogSelect` uses TextEdit to handle text entry and editing and returns `true` and the item number. If there is no edit-text item, `DialogSelect` returns `false`.

· For mouse-downs in an edit text item, `DialogSelect` displays the insertion point caret or selects text as appropriate. `DialogSelect` returns `false` if the edit text item is disabled, and `true` and the item number if it is enabled. (Normally, edit text items should be disabled.)

· For mouse-downs 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.

· For mouse-downs on a disabled item, or 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 item, 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 (for Mac OS 8/9) before calling the 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 control) and pass any other Command-key equivalents to your application's menu handling function.

[3] A major difference between modal alerts and dialogs and movable modal alerts and dialogs 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.

Closing and Disposing of Dialogs

`CloseDialog` closes the dialog's window and removes it from the screen, and frees up the memory occupied by most types of items in the item list and related data structures. It does not release the memory occupied by the dialog object or item list.

`DisposeDialog` closes the dialog's window and deletes it from the window list, and releases the memory occupied by the dialog object, item list, and most types of items. (Handles leading to icons and pictures are not released.)

For modeless dialogs, you might find it more efficient to hide the dialog with `HideWindow` rather than dispose of the dialog. In that way, the dialog will remain available, and in the same location and with the same settings as when it was last used.

Creating Displaying and Handling Window-Modal (Sheet) Alerts and Dialogs

Window-Modal (Sheet) Alerts

Window-modal (sheet) alerts are created using the function `CreateStandardSheet`:

```
OSStatus  CreateStandardSheet(AlertType alertType,CFStringRef error,
                              CFStringRef explanation,
                              const AlertStdCFStringAlertParamRec *param,
                              EventTargetRef notifyTarget, DialogRef *outSheet);
```

`alertType`	The level of the alert. Relevant constants are:
	`kAlertStopAlert`
	`kAlertNoteAlert`
	`kAlertCautionAlert`
	`kAlertPlainAlert`

`error`	The message text.
`explanation`	The informative text.
`param`	A pointer to a **standard CFString alert parameter structure** (see above). `NULL` indicates that none of the features provided by the standard alert structure are required.
`notifyTarget`	The event target to be notified when the sheet is closed.
`outSheet`	On return, the sheet's dialog reference.

The sheet will be invisible when created. A call to `ShowSheetWindow` displays the sheet.

If the sheet has more than one button, your application will need to determine which button was hit by the user. This requires the use of the Carbon event model (see Chapter 17) and the installation of an event handler on the event target. The event target is ordinarily the owner window, in which case you pass a reference to that window in a call to `GetWindowEventTarget` and pass the returned event target reference in the `notifyTarget` parameter of `CreateStandardSheet`. The Carbon event handler you install should respond to the `kEventProcessCommand` Carbon event type and should test for the command IDs `kHICommandOK`, `kHICommandCancel`, and `kHICommandOther` in order to determine which button was hit.

If the sheet has only one button (an OK button), you can simply pass the returned event target reference in the `notifyTarget` parameter (so that the `CreateStandardSheet` call will not fail) and not install a handler. The sheet will be dismissed when the button is hit.

Window-Modal (Sheet) Dialogs

Window-modal (sheet) dialogs, like other dialogs, may be created using `GetNewDialog` or `NewFeaturesDialog`. The window definition ID should be `kWindowSheetProc` (`1088`) and the dialog should be created invisible.

A call to `ShowSheetWindow` displays the sheet. Your application should ensure that only one sheet is displayed in a window at one time.

Events in window-modal (sheet) dialogs may be handled in the same way as for modeless dialogs.

Balloon Help For Dialogs — Mac OS 8/9

Two basic options are available for adding help balloons to dialogs for Mac OS 8/9:

- 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.
- Supplying a window help (`'hwin'`) resource, which will associate help balloons defined in either `'hrct'` resources or `'hdlg'` resources with the dialog's window.[4]

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 dialogs with no titles (see Chaper 4 — Windows, Fig 14). 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 methodolgy is used, the Help Manager automatically tracks the cursor and displays help balloons when the following conditions are met: the dialog has a help item in its `'DITL'` resource; your application calls the Dialog Manager functions `ModalDialog`, or `IsDialogEvent`,; balloon help is enabled.

Figs 13 and 14 at Chapter 4 show `'hrct'` and `'hwin'` resources being created using Resorcerer. Figs 13 and 14 below show a help item and a `'hdlg'` (dialog help) resource being created using Resorcerer.

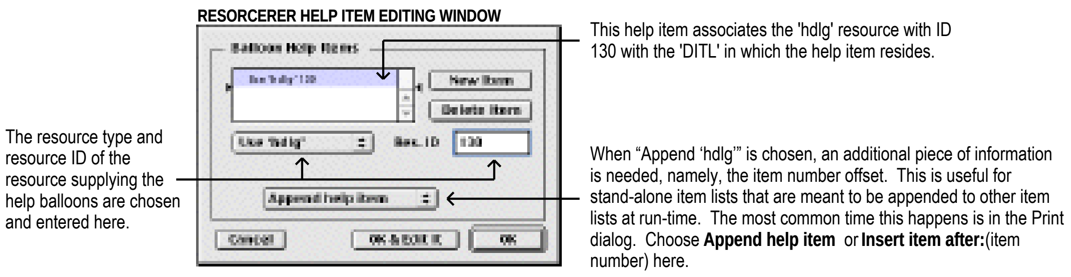
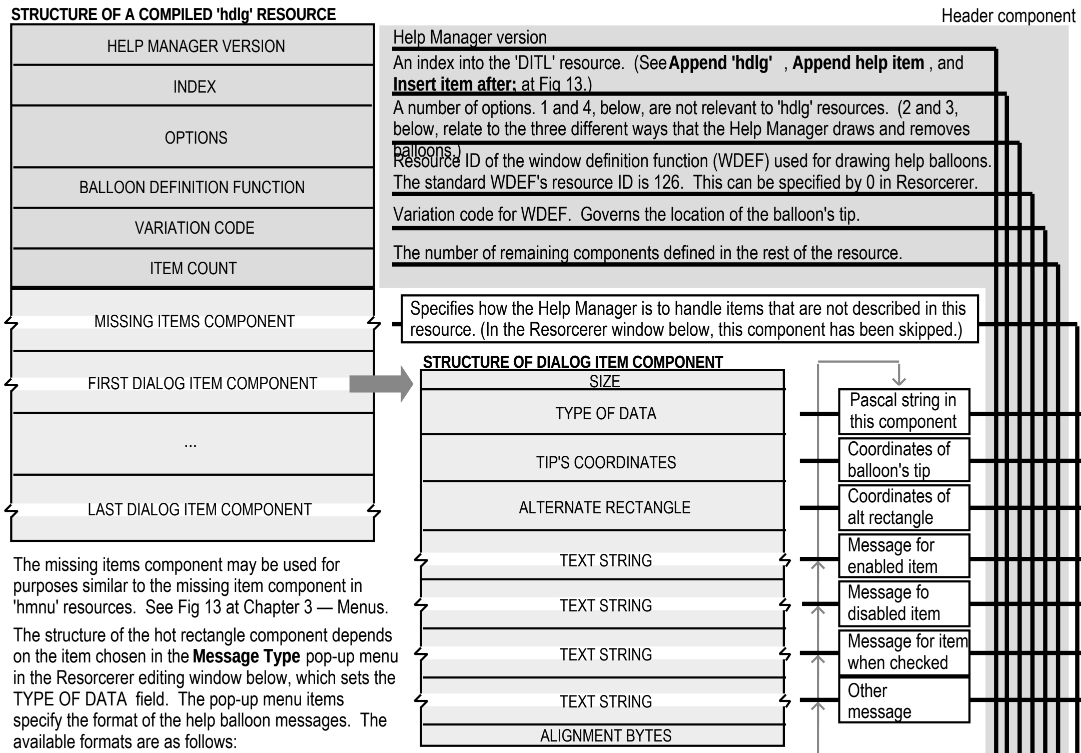

RESORCERER HELP ITEM EDITING WINDOW
Balloon Help Items
Use 'hdlg' 130
New Item
Delete Item
Use 'hdlg'
Res. ID
130
Append help item
Cancel
OK & Edit It
OK
This help item associates the 'hdlg' resource with ID 130 with the 'DITL' in which the help item resides.
The resource type and resource ID of the resource supplying the help balloons are chosen and entered here.
When "Append 'hdlg'" is chosen, an additional piece of information is needed, namely, the item number offset. This is useful for stand-alone item lists that are meant to be appended to other item lists at run-time. The most common time this happens is in the Print dialog. Choose **Append help item** or **Insert item after:**(item number) here.

FIG 13 - CREATING A HELP ITEM USING RESORCERER

[4] `'hrct'` and `'hwin'` resources are described at Chapter 4.



Help Manager version

An index into the 'DITL' resource. (See **Append 'hdlg'** , **Append help item** , and **Insert item after:** at Fig 13.)

A number of options. 1 and 4, below, are not relevant to 'hdlg' resources. (2 and 3, below, relate to the three different ways that the Help Manager draws and removes balloons.)

Resource ID of the window definition function (WDEF) used for drawing help balloons. The standard WDEF's resource ID is 126. This can be specified by 0 in Resorcerer.

Variation code for WDEF. Governs the location of the balloon's tip.

The number of remaining components defined in the rest of the resource.

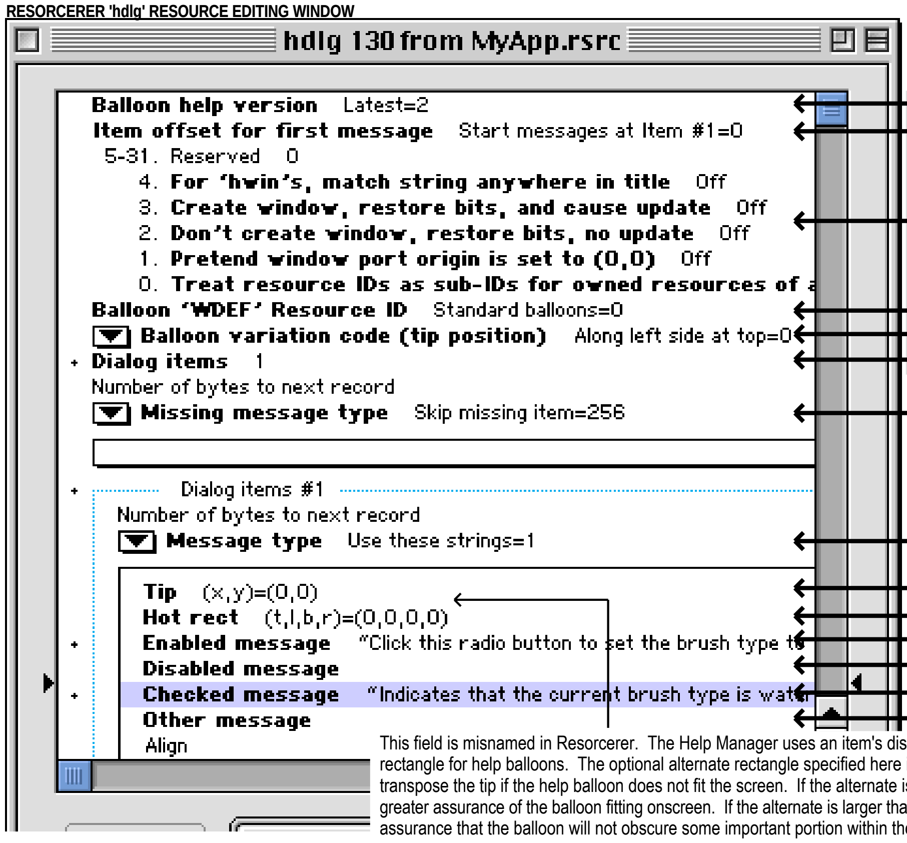
Specifies how the Help Manager is to handle items that are not described in this resource. (In the Resorcerer window below, this component has been skipped.)

The missing items component may be used for purposes similar to the missing item component in 'hmnu' resources. See Fig 13 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 'hrct' 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 from MyApp.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
Balloon 'WDEF' Resource ID   Standard balloons=0
Balloon variation code (tip position)   Along left side at top=0
Dialog items   1
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
    Disabled message
    Checked message   "Indicates that the current brush type is water
    Other message
    Align
```

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 14 - CREATING A 'hdlg' RESOURCE USING RESORCERER

Help Tags For Dialogs — Mac OS X

Balloon help is not available on Mac OS X. On Mac OS X, you should use help tags instead. Help tag creation is addressed at Chapter 25.

Main Dialog Manager Constants, Data Types and Functions

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
```

Standard Item Numbers for OK and Cancel Push Buttons

```
KStdOKItemIndex                 = 1
KStdCancelItemIndex             = 2
```

Resource IDs of Alert Icons

```
kStopIcon                       = 0
kNoteIcon                       = 1
kCautionIcon                    = 2
```

Dialog Item List Manipulation

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

Alert Types

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

Standard Alert Push Button Numbers

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

Alert Default Text

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

Dialog Feature Flags

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

Dialog Font Flags

```
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 in NewFeaturesDialog inProcID Parameter

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

Data Types

```
typedef struct OpaqueDialogPtr *DialogPtr;
typedef DialogPtr DialogRef;
```

Standard Alert Parameter Structure

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

Standard CFStringAlert Alert Paramater Structure

```
struct AlertStdCFStringAlertParamRec
{
  UInt32      version;
  Boolean     movable;
  Boolean     helpButton;
  CFStringRef defaultText;
  CFStringRef cancelText;
  CFStringRef otherText;
  SInt16      defaultButton;
  SInt16      cancelButton;
  UInt16      position;
  OptionBits  flags;
};
typedef struct AlertStdCFStringAlertParamRec AlertStdCFStringAlertParamRec;
typedef AlertStdCFStringAlertParamRec *AlertStdCFStringAlertParamPtr;
```

Functions

Creating Alerts

```
OSErr       StandardAlert(AlertType inAlertType, ConstStr255Param inError,
            ConstStr255Param inExplanation,const AlertStdAlertParamPtr inAlertParam,
            SInt16 *outItemHit);
OSStatus    CreateStandardAlert(AlertType alertType,CFStringRef error,CFStringRef explanation,
            const AlertStdCFStringAlertParamRec *param,DialogRef outAlert);
OSStatus    RunStandardAlert(DialogRef inAlert,ModalFilterUPP filterProc,
            DialogItemIndex *outItemHit);
OSStatus    GetStandardAlertDefaultParams(AlertStdCFStringAlertParamPtr param,UInt32 version);
```

Creating, Closing, and Disposing of Dialogs

```
DialogRef   GetNewDialog(short dialogID,void *dStorage,WindowRef behind);
DialogRef   NewFeaturesDialog(void *inStorage, const Rect *inBoundsRect,
            ConstStr255Param inTitle, Boolean inIsVisible,SInt16 inProcID,WindowRef inBehind,
            Boolean inGoAwayFlag,SInt32 inRefCon,Handle inItemListHandle,UInt32 inFlags);
OSErr       AutoSizeDialog(DialogRef inDialog);
```

```
void        CloseDialog(DialogRef theDialog);
void        DisposeDialog(DialogRef theDialog);
```

Creating Sheets (Mac OS X Only)

```
OSStatus    CreateStandardSheet(AlertType alertType,CFStringRef  error,CFStringRef explanation,
            const AlertStdCFStringAlertParamRec *param,EventTargetRef notifyTarget,
            DialogRef *outSheet);
```

Dialog Object Accessor Functions

```
WindowRef   GetDialogWindow(DialogRef dialog);
TEHandle    GetDialogTextEditHandle(DialogRef dialog);
SInt16      GetDialogKeyboardFocusItem(DialogRef dialog);
SInt16      GetDialogDefaultItem(DialogRef dialog);
OSErr       SetDialogDefaultItem(DialogRef theDialog,DialogItemIndex newItem);
SInt16      GetDialogCancelItem(DialogRef dialog);
OSErr       SetDialogCancelItem(DialogRef theDialog,DialogItemIndex newItem);
```

Utility and Casting Functions

```
void        SetPortDialogPort(DialogRef dialog);
CGrafPtr    GetDialogPort(DialogRef dialog);
DialogRef   GetDialogFromWindow(WindowRef window);
```

Manipulating Items in Alerts and Dialogs

```
void        GetDialogItem(DialogRef theDialog,short itemNo,short *itemType,Handle *item,
            Rect *box);
void        SetDialogItem(DialogRef theDialog,short itemNo,short itemType,Handle item,
            const Rect *box);
OSErr       GetDialogItemAsControl(DialogRef inDialog,SInt16 inItemNo,
            ControlHandle *outControl);
OSErr       MoveDialogItem(DialogRef inDialog,SInt16 inItemNo,SInt16 inHoriz,SInt16 inVert);
OSErr       SizeDialogItem(DialogRef inDialog,SInt16 inItemNo,SInt16 inHeight,SInt16 inWidth);
void        HideDialogItem(DialogRef theDialog,short itemNo);
void        ShowDialogItem(DialogRef theDialog,short itemNo);
short       FindDialogItem(DialogRef theDialog,Point thePt);
void        AppendDITL(DialogRef theDialog,Handle theHandle,DITLMethod theMethod);
void        ShortenDITL(DialogRef theDialog,short numberItems);
OSErr       AppendDialogItemList(DialogRef   dialog,SInt16 ditlID,DITLMethod method);
short       CountDITL(DialogRef the Dialog);
OSStatus    InsertDialogItem (DialogRef theDialog,DialogItemIndex afterItem,
            DialogItemType itemType,Handle itemHandle,const Rect *box);
OSStatus    RemoveDialogItems(DialogRef theDialog,DialogItemIndex itemNo,
            DialogItemIndex amountToRemove,Boolean disposeItemData);
```

Handling Text in Alerts and Dialogs

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

Handling Events in Dialogs

```
void        ModalDialog(ModalFilterUPP modalFilter,short *itemHit);
Boolean     IsDialogEvent(const EventRecord *theEvent);
Boolean     DialogSelect(const EventRecord *theEvent,DialogRef *theDialog,short *itemHit);
void        UpdateDialog(DialogRef theDialog,RgnHandle updateRgn);
void        DrawDialog(DialogRef theDialog);
OSStatus    SetModalDialogEventMask(DialogRef inDialog,EventMask inMask);
OSStatus    GetModalDialogEventMask(DialogRef inDialog,EventMask *outMask);
OSStatus    SetDialogTimeout(DialogRef inDialog,SInt16 inButtonToPress,UInt32 inSecondsToWait);
OSStatus    GetDialogTimeout(DialogRef inDialog,SInt16 *outButtonToPress,
            UInt32 *outSecondsToWait,UInt32 *outSecondsRemaining);
Boolean     StdFilterProc(DialogRef theDialog,EventRecord *event,DialogItemIndex *itemHit);
```

```
OSErr      GetStdFilterProc(ModalFilterUPP *theProc);
OSErr      SetDialogDefaultItem(DialogRef theDialog,DialogItemIndex newItem);
OSErr      SetDialogCancelItem(DialogRef theDialog,DialogItemIndex newItem);
OSErr      SetDialogTracksCursor(DialogRef theDialog,Boolean tracks);
```

Creating and Disposing of Universal Procedure Pointers for Event Filter (Callback) Functions

```
ModalFilterUPP  NewModalFilterUPP(ModalFilterProcPtr userRoutine);
void       DisposeModalFilterUPP(ModalFilterUPP userUPP);
```

Application-Defined (Callback) Function

```
Boolean    myModalFilterFunction(DialogRef theDialog,EventRecord *theEvent,
           DialogItemIndex *itemHit);
```

Relevant Window Manager Functions (Mac OS X Only)

Showing and Hiding Sheets

```
OSStatus   ShowSheetWindow(WindowRef inSheet,WindowRef inParentWindow);
OSStatus   HideSheetWindow(WindowRef inSheet);
OSStatus   GetSheetWindowParent(WindowRef inSheet,WindowRef *outParentWindow);
```

Demonstration Program DialogAndAlerts Listing

```
// *******************************************************************************************
// DialogsAndAlerts.h                                                     CLASSIC EVENT MODEL
// *******************************************************************************************
//
// This program initially opens a small modal dialog which is automatically closed after 10
// seconds, the timeout value having been set by a call to SetDialogTimeout.  The program
// then:
//
// •  Opens a window for the purposes of displaying  advisory text and proving correct window
//    updating and activation/deactivation in the  presence of alerts and dialogs.
//
// •  Allows the user to invoke, via the Demonstration menu, modal and movable modal alerts
//    and dialogs, a modeless dialog and, on Mac OS X, a window-modal alert and dialog
//    (i.e., sheets).
//
// The modal alert box is created programmatically using the StandardAlert function.
//
// The movable modal alert is created programmatically using the StandardAlert function on Mac
// OS 9 and the CreateStandardAlert function on Mac OS X.
//
// The modal dialog contains three checkboxes in one group box, and two pop-up menu buttons in
// another group box.
//
// The movable modal dialog contains four radio buttons in one group box, and a clock control
// and edit text item in another group box.
//
// The modeless dialog contains, amongst other items, an edit text item.
//
// The modal and movable modal alerts and dialogs use an application-defined event filter
// (callback) function.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, and Demonstration pull-down
//    menus, and the pop-up menu buttons (preload, non-purgeable).
//
// •  A 'WIND' resource (purgeable) (initially visible).
//
// •  '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).
//
// •  'STR#' resources (purgeable) containing the message and informative text for the alerts.
//
// •  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
//    movable modal dialog.
//
// •  An 'hrct' resource and associated 'hwin' resource (both purgeable) containing balloon
//    help information for the modeless dialog.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// *******************************************************************************************

// .................................................................................. includes
```

```
#include <Carbon.h>

// .................................................................................................................................................. defines

#define rMenubar                 128
#define mAppleApplication        128
#define  iAbout                  1
#define mFile                    129
#define  iClose                  4
#define  iQuit                   12
#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  iWindowModalAlert       7
#define  iWindowModalDialog      8
#define mFont                    132
#define rWindow                  128
#define rSplash                  128
#define rModalDialog             129
#define  iGridSnap               4
#define  iShowGrid               5
#define  iShowRulers             6
#define  iFont                   11
#define  iSound                  12
#define rMovableModalDialog      130
#define  iCharcoal               7
#define  iOilPaint               8
#define  iPencil                 9
#define  iChalk                  10
#define  iClockOne               12
#define rModelessDialog          131
#define  iEditTextSearchModeless 2
#define rSheetDialog             132
#define  iEditTextSheetDialog    2
#define rAlertStrings            128
#define  sModalMessage           1
#define  sModalInformative       2
#define  sMovableMessage         3
#define  sMovableInformative     4
#define rSheetStrings            132
#define  sAlertSheetMessage      1
#define  sAlertSheetInformative  2
#define rPixelPattern            128
#define kSearchModeless          1
#define kSheetDialog             2

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

#define MAX_UINT32               0xFFFFFFFF

// .......................................................................................................................... function prototypes

void    main                        (void);
void    doPreliminaries             (void);
OSErr   quitAppEventHandler         (AppleEvent *,AppleEvent *,SInt32);
void    eventLoop                   (void);
void    doIdle                      (void);
void    doEvents                      (EventRecord *);
void    doMouseDown                   (EventRecord *);
void    doKeyDown                     (EventRecord *);
void    doUpdate                      (EventRecord *);
void    doUpdateDocument              (WindowRef);
void    doActivate                    (EventRecord *);
void    doActivateDocument            (WindowRef,Boolean);
void    doActivateDialogs             (EventRecord *,Boolean);
void    doOSEvent                     (EventRecord *);
void    doAdjustMenus                 (void);
void    doMenuChoice                  (SInt32);
void    doEditMenu                    (MenuItemIndex);
void    doDemonstrationMenu           (MenuItemIndex);
void    doExplicitlyDeactivateDocument (void);
Boolean doModalAlerts                 (Boolean);
Boolean doMovableModalAlertOnX        (void);
Boolean doModalDialog                 (void);
Boolean doMovableModalDialog          (void);
Boolean doCreateOrShowModelessDialog  (void);
void    doInContent                   (EventRecord *);
void    doButtonHitInSearchModeless   (void);
void    doHideModelessDialog          (WindowRef);
Boolean eventFilter                   (DialogRef,EventRecord *,SInt16 *);
void    doPopupMenuChoice             (ControlRef,SInt16);
void    doDrawMessage                 (WindowRef,Boolean);
void    doCopyPString                 (Str255,Str255);

Boolean doSheetAlert                  (void);
Boolean doSheetDialog                 (void);
void    doButtonHitInSheetDialog      (void);

void    helpTagsModal                 (DialogRef);
void    helpTagsMovableModal          (DialogRef);
void    helpTagsModeless              (DialogRef);

// *******************************************************************************************
// DialogsAndAlerts.c
// *******************************************************************************************

// .................................................................................. includes

#include "DialogsAndAlerts.h"

// .......................................................................... global variables

Boolean        gRunningOnX        = false;
ModalFilterUPP gEventFilterUPP;
Str255         gCurrentString;
WindowRef      gWindowRef;
SInt32         gSleepTime;
Boolean        gDone;
Boolean        gGridSnap          = kControlCheckBoxUncheckedValue;
Boolean        gShowGrid          = kControlCheckBoxUncheckedValue;
Boolean        gShowRule          = kControlCheckBoxUncheckedValue;
SInt16         gBrushType         = iCharcoal;
DialogRef      gModelessDialogRef = NULL;

// *************************************************************************************** main

void  main(void)
{
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;
  DialogRef     dialogRef;
  SInt16        itemHit;

  // ........................................................................ do preliminaries
  doPreliminaries();

  // ................................................................ set up menu bar and menus

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
      DisableMenuItem(menuRef,0);
    }

    gRunningOnX = true;
  }

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

  dialogRef = GetNewDialog(rSplash,NULL,(WindowRef) -1);
  SetDialogTimeout(dialogRef,kStdOkItemIndex,10);

  do
  {
    ModalDialog(NULL,&itemHit);
  } while(itemHit != kStdOkItemIndex);

  DisposeDialog(dialogRef);

  // ............ create universal procedure pointer for event filter function

  gEventFilterUPP = NewModalFilterUPP((ModalFilterProcPtr) eventFilter);

  // ........................................ initial advisory text for window header

  doCopyPString("\pBalloon (OS 8/9) and Help tag (OS X) help is available",gCurrentString);

  // ........................................................ open a window, set font size

  if(!(gWindowRef = GetNewCWindow(rWindow,NULL,(WindowRef)-1)))
    ExitToShell();

  SetPortWindowPort(gWindowRef);
  if(!gRunningOnX)
    TextSize(10);

  // ................................................................... enter eventLoop

  eventLoop();
}

// ************************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  OSErr osError;

  MoreMasterPointers(192);
  InitCursor();
  FlushEvents(everyEvent,0);

  osError = AEInstallEventHandler(kCoreEventClass,kAEQuitApplication,
                            NewAEEventHandlerUPP((AEEventHandlerProcPtr) quitAppEventHandler),
                            0L,false);
  if(osError != noErr)
    ExitToShell();
}

// ************************************************************************ doQuitAppEvent

OSErr  quitAppEventHandler(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
  OSErr    osError;
  DescType returnedType;
  Size     actualSize;

  osError = AEGetAttributePtr(appEvent,keyMissedKeywordAttr,typeWildCard,&returnedType,NULL,0,
                              &actualSize);

  if(osError == errAEDescNotFound)
  {
    gDone = true;
    osError = noErr;
  }
  else if(osError == noErr)
    osError = errAEParamMissed;

  return osError;
}

// ***************************************************************************** eventLoop

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

  gSleepTime = MAX_UINT32;
  gDone = false;

  while(!gDone)
  {
    gotEvent = WaitNextEvent(everyEvent,&eventStructure,gSleepTime,NULL);
    if(gotEvent)
      doEvents(&eventStructure);
    else
    {
      if(eventStructure.what == nullEvent)
        if(!gRunningOnX)
          doIdle();
    }
  }
}

// ******************************************************************************** doIdle

void  doIdle(void)
{
  if(FrontWindow() == GetDialogWindow(gModelessDialogRef))
    IdleControls(GetDialogWindow(gModelessDialogRef));
}

// ****************************************************************************** doEvents

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

    case keyDown:
      doKeyDown(eventStrucPtr);
      break;

    case autoKey:
      if((eventStrucPtr->modifiers & cmdKey) == 0)
        doKeyDown(eventStrucPtr);
      break;

    case updateEvt:
      doUpdate(eventStrucPtr);
      break;

    case activateEvt:
      doActivate(eventStrucPtr);
      break;

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

// ****************************************************************************** doMouseDown

void  doMouseDown(EventRecord *eventStrucPtr)
{
  WindowRef      windowRef;
  WindowPartCode partCode;

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

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

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

    case inDrag:
      DragWindow(windowRef,eventStrucPtr->where,NULL);
      break;

    case inGoAway:
      if(TrackGoAway(windowRef,eventStrucPtr->where))
      {
        if(GetWindowKind(windowRef) == kDialogWindowKind)
        {
          doHideModelessDialog(windowRef);
          doCopyPString("\pBalloon (OS 8/9) and Help tag (OS X) help is available",
                        gCurrentString);
        }
      }
      break;
  }
}
```

```
// ****************************************************************************** doKeyDown

void  doKeyDown(EventRecord *eventStrucPtr)
{
  WindowRef  windowRef;
  SInt8      charCode;
  SInt32     windowRefCon;
  SInt16     itemHit;
  ControlRef controlRef;
  UInt32     finalTicks;
  DialogRef  dialogRef;

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

  if(!(IsDialogEvent(eventStrucPtr)))
  {
    if((eventStrucPtr->modifiers & cmdKey) != 0)
    {
      doAdjustMenus();
      doMenuChoice(MenuEvent(eventStrucPtr));
    }
  }
  else
  {
    windowRefCon = GetWRefCon(windowRef);
    if(windowRefCon == kSearchModeless || windowRefCon == kSheetDialog)
    {
      if((charCode == kReturn) || (charCode == kEnter))
      {
        GetDialogItemAsControl(GetDialogFromWindow(windowRef),kStdOkItemIndex,&controlRef);
        HiliteControl(controlRef,kControlButtonPart);
        Delay(8,&finalTicks);
        HiliteControl(controlRef,kControlEntireControl);
        if(windowRefCon == kSearchModeless)
          doButtonHitInSearchModeless();
        else if(windowRefCon == kSheetDialog)
          doButtonHitInSheetDialog();
        return;
      }

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

        return;
      }

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

// ******************************************************************************* doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
  WindowRef windowRef;
  DialogRef dialogRef;
  SInt16    itemHit;

  if(!(IsDialogEvent(eventStrucPtr)))
  {
    windowRef = (WindowRef) eventStrucPtr->message;
    doUpdateDocument(windowRef);
  }
  else
    DialogSelect(eventStrucPtr,&dialogRef,&itemHit);
}

// *************************************************************************** doUpdateDocument

void  doUpdateDocument(WindowRef windowRef)
{
  GrafPtr      oldPort;
  PixPatHandle pixpatHdl;
  Rect         portRect;

  BeginUpdate(windowRef);

  GetPort(&oldPort);
  SetPortWindowPort(windowRef);

  pixpatHdl = GetPixPat(rPixelPattern);
  GetWindowPortBounds(windowRef,&portRect);
  FillCRect(&portRect,pixpatHdl);
  DisposePixPat(pixpatHdl);
  doDrawMessage(windowRef,windowRef == FrontWindow());

  SetPort(oldPort);

  EndUpdate(windowRef);
}

// ********************************************************************************* doActivate

void  doActivate(EventRecord *eventStrucPtr)
{
  Boolean   becomingActive;
  WindowRef windowRef;

  becomingActive = (eventStrucPtr->modifiers & activeFlag) == activeFlag;

  if(!(IsDialogEvent(eventStrucPtr)))
  {
    windowRef = (WindowRef) eventStrucPtr->message;
    doActivateDocument(windowRef,becomingActive);
  }
  else
    doActivateDialogs(eventStrucPtr,becomingActive);
}

// ************************************************************************* doActivateDocument

void  doActivateDocument(WindowRef windowRef,Boolean becomingActive)
{
  if(becomingActive)
    doAdjustMenus();

  doDrawMessage(windowRef,becomingActive);
}

// ******************************************************************* doActivateModelessDialog

void  doActivateDialogs(EventRecord *eventStrucPtr,Boolean becomingActive)
{
  DialogRef dialogRef;
  SInt16    windowRefCon;
  SInt16    itemHit;

  DialogSelect(eventStrucPtr,&dialogRef,&itemHit);

  windowRefCon = GetWRefCon(GetDialogWindow(dialogRef));

  if(becomingActive)
  {
    doAdjustMenus();
    if(windowRefCon == kSearchModeless || windowRefCon == kSheetDialog)
      gSleepTime = GetCaretTime();
  }
  else
  {
    if(windowRefCon == kSearchModeless || windowRefCon == kSheetDialog)
      gSleepTime = MAX_UINT32;
  }
}

// ******************************************************************************** doOSEvent

void  doOSEvent(EventRecord *eventStrucPtr)
{
  switch((eventStrucPtr->message >> 24) & 0x000000FF)
  {
    case suspendResumeMessage:
      if((eventStrucPtr->message & resumeFlag) == 1)
        SetThemeCursor(kThemeArrowCursor);
      break;
  }
}

// **************************************************************************** doAdjustMenus

void  doAdjustMenus(void)
{
  WindowRef windowRef;
  MenuRef   menuRef;
  SInt32    windowRefCon;

  windowRef = FrontWindow();

  if(GetWindowKind(windowRef) == kApplicationWindowKind)
  {
    menuRef = GetMenuRef(mFile);
    DisableMenuItem(menuRef,iClose);
    menuRef = GetMenuRef(mEdit);
    DisableMenuItem(menuRef,0);
    menuRef = GetMenuRef(mDemonstration);
    EnableMenuItem(menuRef,iModeless);
    if(gRunningOnX)
    {
      if(IsWindowCollapsed(gWindowRef))
      {
        DisableMenuItem(menuRef,iWindowModalDialog);
        DisableMenuItem(menuRef,iWindowModalAlert);
      }
      else
      {
        EnableMenuItem(menuRef,iWindowModalDialog);
        EnableMenuItem(menuRef,iWindowModalAlert);
      }
    }
  }
  else if(GetWindowKind(windowRef) == kDialogWindowKind)
  {
    windowRefCon = GetWRefCon(windowRef);
    if(windowRefCon == kSearchModeless)
    {
      menuRef = GetMenuRef(mFile);
      EnableMenuItem(menuRef,iClose);
      menuRef = GetMenuRef(mEdit);
      EnableMenuItem(menuRef,0);
      menuRef = GetMenuRef(mDemonstration);
      DisableMenuItem(menuRef,iModeless);
    }
    else if(windowRefCon == kSheetDialog)
    {
      menuRef = GetMenuRef(mFile);
      DisableMenuItem(menuRef,iClose);
      menuRef = GetMenuRef(mEdit);
      EnableMenuItem(menuRef,0);
      menuRef = GetMenuRef(mDemonstration);
      EnableMenuItem(menuRef,iModeless);
      DisableMenuItem(menuRef,iWindowModalAlert);
    }
    else
    {
      menuRef = GetMenuRef(mFile);
      DisableMenuItem(menuRef,iClose);
      menuRef = GetMenuRef(mEdit);
      DisableMenuItem(menuRef,0);
      menuRef = GetMenuRef(mDemonstration);
      EnableMenuItem(menuRef,iModeless);
    }
  }

  DrawMenuBar();
}

// ****************************************************************************** doMenuChoice

void  doMenuChoice(SInt32 menuChoice)
{
  MenuID        menuID;
  MenuItemIndex menuItem;
  SInt16        windowRefCon;

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

  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mAppleApplication:
      if(menuItem == iAbout)
        SysBeep(10);
      break;

    case mFile:
      if(menuItem == iQuit)
        gDone = true;
      else if(menuItem == iClose)
      {
        if(GetWindowKind(FrontWindow()) == kDialogWindowKind)
        {
          windowRefCon = GetWRefCon(FrontWindow());
          if(windowRefCon == kSearchModeless)
            doHideModelessDialog(GetDialogWindow(gModelessDialogRef));
        }
      }
      break;

    case mEdit:
      doEditMenu(menuItem);
      break;

    case mDemonstration:
      doDemonstrationMenu(menuItem);
      break;
  }

  HiliteMenu(0);
}

// ******************************************************************************** doEditMenu

void  doEditMenu(MenuItemIndex menuItem)
{
  WindowRef windowRef;
  SInt16    windowRefCon;
  DialogRef dialogRef;

  windowRef = FrontWindow();

  if(GetWindowKind(FrontWindow()) == kDialogWindowKind)
  {
    windowRefCon = GetWRefCon(windowRef);
    if(windowRefCon == kSearchModeless || windowRefCon == kSheetDialog)
    {
      dialogRef = GetDialogFromWindow(windowRef);

      switch(menuItem)
      {
        case iCut:
          DialogCut(dialogRef);
          break;

        case iCopy:
          DialogCopy(dialogRef);
          break;

        case iPaste:
          DialogPaste(dialogRef);
          break;

        case iClear:
          DialogDelete(dialogRef);
          break;
      }
    }
  }
}

// *********************************************************************** doDemonstrationMenu

void  doDemonstrationMenu(MenuItemIndex menuItem)
{
  switch(menuItem)
  {
    case iModalAlert:
      if(!doModalAlerts(false))
      {
        SysBeep(10);
        ExitToShell();
      }
      break;

    case iMovableAlert:
      if(gRunningOnX)
      {
        if(!doMovableModalAlertOnX())
        {
          SysBeep(10);
          ExitToShell();
        }
      }
      else if(!doModalAlerts(true))
      {
        SysBeep(10);
        ExitToShell();
      }
      break;

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

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

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

    case iWindowModalAlert:
      if(!doSheetAlert())
      {
        SysBeep(10);
        ExitToShell();
      }
      break;

    case iWindowModalDialog:
      if(!doSheetDialog())
      {
        SysBeep(10);
        ExitToShell();
      }
      break;
  }
}

// ************************************************************ doExplicitlyDeactivateDocument

void  doExplicitlyDeactivateDocument(void)
{
  if(FrontWindow() && (GetWindowKind(FrontWindow()) != kDialogWindowKind))
    doActivateDocument(FrontWindow(),false);
}

// ***************************************************************************** doModalAlerts

Boolean  doModalAlerts(Boolean movable)
{
  AlertStdAlertParamRec paramRec;
  Str255                messageText, informativeText;
  Str255                otherText = "\pOther";
  OSErr                 osError;
  DialogItemIndex       itemHit;

  doExplicitlyDeactivateDocument();

  paramRec.movable       = movable;
  paramRec.helpButton    = true;
  paramRec.filterProc    = gEventFilterUPP;
  paramRec.defaultText   = (StringPtr) kAlertDefaultOKText;
  paramRec.cancelText    = (StringPtr) kAlertDefaultCancelText;
  paramRec.otherText     = (StringPtr) &otherText;
  paramRec.defaultButton = kAlertStdAlertOKButton;
  paramRec.cancelButton  = kAlertStdAlertCancelButton;
  paramRec.position      = kWindowDefaultPosition;

  if(!movable)
    GetIndString(messageText,rAlertStrings,sModalMessage);
  else
    GetIndString(messageText,rAlertStrings,sMovableMessage);
  GetIndString(informativeText,rAlertStrings,sModalInformative);

  osError = StandardAlert(kAlertStopAlert,messageText,informativeText,&paramRec,&itemHit);
  if(osError == noErr)
  {
    if(itemHit == kAlertStdAlertOKButton)
      doCopyPString("\pOK Button hit",gCurrentString);
    else if (itemHit == kAlertStdAlertCancelButton)
      doCopyPString("\pCancel Button hit",gCurrentString);
    else if (itemHit == kAlertStdAlertOtherButton)
      doCopyPString("\pOther Button hit",gCurrentString);
    else if (itemHit == kAlertStdAlertHelpButton)
      doCopyPString("\pHelp Button hit",gCurrentString);
  }

  return (osError == noErr);
}

// ****************************************************************** doMovableModalAlertOnX

Boolean  doMovableModalAlertOnX(void)
{
  AlertStdCFStringAlertParamRec paramRec;
  Str255                        messageText, informativeText;
  CFStringRef                   messageTextCF, informativeTextCF;
  OSErr                         osError;
  DialogRef                     dialogRef;
  DialogItemIndex               itemHit;

  doExplicitlyDeactivateDocument();

  GetStandardAlertDefaultParams(&paramRec,kStdCFStringAlertVersionOne);
  paramRec.movable      = true;
  paramRec.helpButton   = true;
  paramRec.cancelButton = kAlertStdAlertCancelButton;
  paramRec.cancelText   = CFSTR("Cancel");
  paramRec.otherText    = CFSTR("Other");

  GetIndString(messageText,rAlertStrings,sMovableMessage);
  GetIndString(informativeText,rAlertStrings,sMovableInformative);
  messageTextCF = CFStringCreateWithPascalString(NULL,messageText,
                                                 CFStringGetSystemEncoding());
  informativeTextCF = CFStringCreateWithPascalString(NULL,informativeText,
                                                     CFStringGetSystemEncoding());

  osError = CreateStandardAlert(kAlertCautionAlert,messageTextCF,informativeTextCF,&paramRec,
                                &dialogRef);
  if(osError == noErr)
  {
    osError = RunStandardAlert(dialogRef,NULL,&itemHit);
    if(osError == noErr)
    {
      if(itemHit == kAlertStdAlertOKButton)
        doCopyPString("\pOK Button hit",gCurrentString);
      else if (itemHit == kAlertStdAlertCancelButton)
        doCopyPString("\pCancel Button hit",gCurrentString);
      else if (itemHit == kAlertStdAlertOtherButton)
        doCopyPString("\pOther Button hit",gCurrentString);
      else if (itemHit == kAlertStdAlertHelpButton)
        doCopyPString("\pHelp Button hit",gCurrentString);
    }
  }

  if(messageTextCF != NULL)
    CFRelease(messageTextCF);
  if(informativeTextCF != NULL)
    CFRelease(informativeTextCF);

  return (osError == noErr);
}

// ************************************************************************* doModalDialog

Boolean  doModalDialog(void)
{
  DialogRef  dialogRef;
  ControlRef controlRef;
  OSStatus   osError;
  MenuRef    menuRef;
  SInt16     numberOfItems, itemHit, controlValue;

  doExplicitlyDeactivateDocument();

  if(!(dialogRef = GetNewDialog(rModalDialog,NULL,(WindowRef) -1)))
    return false;

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

  GetDialogItemAsControl(dialogRef,iGridSnap,&controlRef);
  SetControlValue(controlRef,gGridSnap);
  GetDialogItemAsControl(dialogRef,iShowGrid,&controlRef);
  SetControlValue(controlRef,gShowGrid);
  GetDialogItemAsControl(dialogRef,iShowRulers,&controlRef);
  SetControlValue(controlRef,gShowRule);

  menuRef = NewMenu(mFont,NULL);

  if((osError = CreateStandardFontMenu(menuRef,0,0,0,NULL)) == noErr)
  {
    GetDialogItemAsControl(dialogRef,iFont,&controlRef);
    SetControlMinimum(controlRef,1);
    numberOfItems = CountMenuItems(menuRef);
    SetControlMaximum(controlRef,numberOfItems);
    SetControlData(controlRef,kControlEntireControl,kControlPopupButtonMenuRefTag,
                   sizeof(menuRef),&menuRef);
  }
  else
    return false;

  if(gRunningOnX)
    helpTagsModal(dialogRef);

  ShowWindow(GetDialogWindow(dialogRef));

  do
  {
    ModalDialog(gEventFilterUPP,&itemHit);

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

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

  DisposeDialog(dialogRef);

  doCopyPString("\pBalloon (OS 8/9) and Help tag (OS X) help is available",gCurrentString);

  return true;
}

// ******************************************************************** doMovableModalDialog

Boolean  doMovableModalDialog(void)
{
  DialogRef  dialogRef;
  ControlRef controlRef;
  SInt16     oldBrushType, itemHit, a;

  doExplicitlyDeactivateDocument();

  if(!(dialogRef = GetNewDialog(rMovableModalDialog,NULL,(WindowRef) -1)))
    return false;

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

  GetDialogItemAsControl(dialogRef,gBrushType,&controlRef);
  SetControlValue(controlRef,kControlRadioButtonCheckedValue);

  GetDialogItemAsControl(dialogRef,iClockOne,&controlRef);
  SetKeyboardFocus(GetDialogWindow(dialogRef),controlRef,kControlClockPart);

  oldBrushType = gBrushType;

  if(gRunningOnX)
    helpTagsMovableModal(dialogRef);

  ShowWindow(GetDialogWindow(dialogRef));

  do
  {
    ModalDialog(gEventFilterUPP,&itemHit);

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

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

  if(itemHit == kStdCancelItemIndex)
    gBrushType = oldBrushType;

  DisposeDialog(dialogRef);

  return true;
}

// ************************************************************** doCreateOrShowModelessDialog

Boolean  doCreateOrShowModelessDialog(void)
{
  ControlRef controlRef;
  Str255     stringData = "\pwicked googly";
  MenuRef    menuRef;

  if(gModelessDialogRef == NULL)
  {
    if(!(gModelessDialogRef = GetNewDialog(rModelessDialog,NULL,(WindowRef) -1)))
      return false;

    SetWRefCon(GetDialogWindow(gModelessDialogRef),(SInt32) kSearchModeless);

    SetDialogDefaultItem(gModelessDialogRef,kStdOkItemIndex);

    GetDialogItemAsControl(gModelessDialogRef,iEditTextSearchModeless,&controlRef);
    SetDialogItemText((Handle) controlRef,stringData);
    SelectDialogItemText(gModelessDialogRef,iEditTextSearchModeless,0,32767);

    if(gRunningOnX)
      helpTagsModeless(gModelessDialogRef);

    ShowWindow(GetDialogWindow(gModelessDialogRef));
  }
  else
  {
    ShowWindow(GetDialogWindow(gModelessDialogRef));
    SelectWindow(GetDialogWindow(gModelessDialogRef));
  }

  if(gRunningOnX)
  {
    menuRef = GetMenuRef(mFile);
    EnableMenuItem(menuRef,0);
  }

  return true;
}

// ******************************************************************************* doInContent

void  doInContent(EventRecord *eventStrucPtr)
{
  WindowRef windowRef;
  SInt32    windowRefCon;
  DialogRef dialogRef;
  SInt16    itemHit;

  windowRef = FrontWindow();

  if(!(IsDialogEvent(eventStrucPtr)))
  {
    // Handle clicks in document window content region here.
  }
  else
  {
    windowRefCon = GetWRefCon(windowRef);
    if(windowRefCon == kSearchModeless)
    {
      if(DialogSelect(eventStrucPtr,&dialogRef,&itemHit))
        if(itemHit == kStdOkItemIndex)
          doButtonHitInSearchModeless();
    }
    else if(windowRefCon == kSheetDialog)
    {
      if(DialogSelect(eventStrucPtr,&dialogRef,&itemHit))
        if(itemHit == kStdOkItemIndex)
          doButtonHitInSheetDialog();
    }
  }
}

// ************************************************************ doButtonHitInSearchModeless

void  doButtonHitInSearchModeless(void)
{
  ControlRef controlRef;
  GrafPtr    oldPort;

  GetDialogItemAsControl(gModelessDialogRef,iEditTextSearchModeless,&controlRef);
  GetDialogItemText((Handle) controlRef,gCurrentString);

  GetPort(&oldPort);
  SetPortWindowPort(gWindowRef);
  doDrawMessage(gWindowRef,false);
  SetPort(oldPort);
}

// ******************************************************************* doHideModelessDialog

void  doHideModelessDialog(WindowRef windowRef)
{
  SInt16  windowRefCon;
  MenuRef menuRef;

  if(gRunningOnX)
    BringToFront(gWindowRef);

  HideWindow(windowRef);

  windowRefCon = GetWRefCon(windowRef);
  if(windowRefCon == kSearchModeless)
    gSleepTime = MAX_UINT32;

  if(gRunningOnX)
  {
    menuRef = GetMenuRef(mFile);
    DisableMenuItem(menuRef,0);
  }
}

// ***************************************************************************** eventFilter

Boolean  eventFilter(DialogRef dialogRef,EventRecord *eventStrucPtr,SInt16 *itemHit)
{
  Boolean handledEvent;
  GrafPtr oldPort;

  handledEvent = false;
  if((eventStrucPtr->what == updateEvt) &&
     ((WindowRef) eventStrucPtr->message != GetDialogWindow(dialogRef)))
  {
    if(!gRunningOnX)
      doUpdate(eventStrucPtr);
  }
  else if((eventStrucPtr->what == autoKey) && ((eventStrucPtr->modifiers & cmdKey) != 0))
  {
    handledEvent = true;
    return handledEvent;
  }
  else
  {
    GetPort(&oldPort);
    SetPortDialogPort(dialogRef);

    handledEvent = StdFilterProc(dialogRef,eventStrucPtr,itemHit);

    SetPort(oldPort);
  }

  return handledEvent;
}

// ********************************************************************** doPopupMenuChoice

void  doPopupMenuChoice(ControlRef controlRef,SInt16 controlValue)
{
  MenuRef menuRef;
  Size    actualSize;
  Str255  itemName;
  GrafPtr oldPort;

  GetControlData(controlRef,kControlEntireControl,kControlPopupButtonMenuHandleTag,
                 sizeof(menuRef),&menuRef,&actualSize);
  GetMenuItemText(menuRef,controlValue,itemName);
  doCopyPString(itemName,gCurrentString);

  GetPort(&oldPort);
  SetPortWindowPort(gWindowRef);
  doDrawMessage(gWindowRef,false);
  SetPort(oldPort);
}

// ************************************************************************** doDrawMessage

void  doDrawMessage(WindowRef windowRef,Boolean inState)
{
  Rect        portRect, headerRect;
  CFStringRef stringRef;
  Rect        textBoxRect;

  if(windowRef == gWindowRef)
  {
    SetPortWindowPort(windowRef);
    GetWindowPortBounds(windowRef,&portRect);
    SetRect(&headerRect,portRect.left - 1,portRect.bottom - 26,portRect.right + 1,
            portRect.bottom + 1);
    DrawThemePlacard(&headerRect,inState);

    if(inState == kThemeStateActive)
      TextMode(srcOr);
    else
      TextMode(grayishTextOr);

    stringRef = CFStringCreateWithPascalString(NULL,gCurrentString,
                                               CFStringGetSystemEncoding());
    SetRect(&textBoxRect,portRect.left,portRect.bottom - 19,portRect.right,
              portRect.bottom - 4);
    DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,true,&textBoxRect,teJustCenter,NULL);
    if(stringRef != NULL)
      CFRelease(stringRef);

    TextMode(srcOr);
  }
}

// *************************************************************************** doCopyPString

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

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

// ******************************************************************************************
// Sheets.c
// ******************************************************************************************

// .......................................................................................... includes

#include "DialogsAndAlerts.h"

// .................................................................................. global variables

WindowRef gSheetDialogWindowRef = NULL;

extern WindowRef gWindowRef;
extern Boolean   gRunningOnX;
extern Str255    gCurrentString;

// *************************************************************************** doSheetAlert

Boolean  doSheetAlert(void)
{
  AlertStdCFStringAlertParamRec paramRec;
  Str255                        messageText, informativeText;
  CFStringRef                   messageTextCF, informativeTextCF;
  OSStatus                      osError;
  DialogRef                     dialogRef;
  MenuRef                       menuRef;

  GetStandardAlertDefaultParams(&paramRec,kStdCFStringAlertVersionOne);

  GetIndString(messageText,rSheetStrings,sAlertSheetMessage);
  GetIndString(informativeText,rSheetStrings,sAlertSheetInformative);
  messageTextCF = CFStringCreateWithPascalString(NULL,messageText,
                                                 CFStringGetSystemEncoding());
  informativeTextCF = CFStringCreateWithPascalString(NULL,informativeText,
                                                     CFStringGetSystemEncoding());

  osError = CreateStandardSheet(kAlertCautionAlert,messageTextCF,informativeTextCF,&paramRec,
                                GetWindowEventTarget(gWindowRef),&dialogRef);
  if(osError == noErr)
    osError = ShowSheetWindow(GetDialogWindow(dialogRef),gWindowRef);

  CFRelease(messageTextCF);
  CFRelease(informativeTextCF);

  menuRef = GetMenuRef(mDemonstration);
  if(menuRef != NULL)
  {
    DisableMenuItem(menuRef,iWindowModalDialog);
    DisableMenuItem(menuRef,iWindowModalAlert);
  }
  return (osError == noErr);
}

// ***************************************************************************** doSheetDialog

Boolean  doSheetDialog(void)
{
  DialogRef  dialogRef;
  ControlRef controlRef;
  Str255     stringData = "\pBradman";
  OSStatus   osError = noErr;
  MenuRef    menuRef;

  if(!(dialogRef = GetNewDialog(rSheetDialog,NULL,(WindowRef) -1)))
    return false;

  SetWRefCon(GetDialogWindow(dialogRef),(SInt32) kSheetDialog);

  SetDialogDefaultItem(dialogRef,kStdOkItemIndex);

  GetDialogItemAsControl(dialogRef,iEditTextSheetDialog,&controlRef);
  SetDialogItemText((Handle) controlRef,stringData);
  SelectDialogItemText(dialogRef,iEditTextSheetDialog,0,32767);

  gSheetDialogWindowRef = GetDialogWindow(dialogRef);
  osError = ShowSheetWindow(gSheetDialogWindowRef,gWindowRef);

  menuRef = GetMenuRef(mDemonstration);
  if(menuRef != NULL)
    DisableMenuItem(menuRef,iWindowModalDialog);

  return (osError == noErr);
}

// ****************************************************************** doButtonHitInSheetDialog

void  doButtonHitInSheetDialog(void)
{
  DialogRef  dialogRef;
  ControlRef controlRef;
  GrafPtr    oldPort;

  dialogRef = GetDialogFromWindow(gSheetDialogWindowRef);

  GetDialogItemAsControl(dialogRef,iEditTextSheetDialog,&controlRef);
  GetDialogItemText((Handle) controlRef,gCurrentString);

  HideSheetWindow(gSheetDialogWindowRef);
  DisposeDialog(dialogRef);
  gSheetDialogWindowRef = NULL;

  GetPort(&oldPort);
  SetPortWindowPort(gWindowRef);
  doDrawMessage(gWindowRef,true);
  SetPort(oldPort);
}

// *******************************************************************************************
// HelpTags.c
// *******************************************************************************************

// ........................................................................................ includes

#include "DialogsAndAlerts.h"
#include <string.h>

// ***************************************************************************** helpTagsModal
```

```
void  helpTagsModal(DialogRef dialogRef)
{
  HMHelpContentRec helpContent;
  SInt16           a;
  static SInt16    itemNumber[7] = { 1,2,3,7,8,10,11 };
  ControlRef       controlRef;

  memset(&helpContent,0,sizeof(helpContent));

  HMSetTagDelay(500);
  HMSetHelpTagsDisplayed(true);

  helpContent.version = kMacHelpVersion;
  helpContent.tagSide = kHMOutsideTopCenterAligned;
  helpContent.content[kHMMinimumContentIndex].contentType = kHMStringResContent;
  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmResID = 129;

  for(a = 1;a <= 7; a++)
  {
    helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = a;
    GetDialogItemAsControl(dialogRef,itemNumber[a - 1],&controlRef);
    HMSetControlHelpContent(controlRef,&helpContent);
  }
}

// ******************************************************************** helpTagsMovableModal

void  helpTagsMovableModal(DialogRef dialogRef)
{
  HMHelpContentRec helpContent;
  SInt16           a;
  static SInt16    itemNumber[9] = { 1,2,3,4,6,11,12,13,14 };
  ControlRef       controlRef;

  memset(&helpContent,0,sizeof(helpContent));

  HMSetTagDelay(500);
  HMSetHelpTagsDisplayed(true);

  helpContent.version = kMacHelpVersion;
  helpContent.tagSide = kHMOutsideTopCenterAligned;
  helpContent.content[kHMMinimumContentIndex].contentType = kHMStringResContent;
  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmResID = 130;

  for(a = 1;a <= 9; a++)
  {
    helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = a;
    GetDialogItemAsControl(dialogRef,itemNumber[a - 1],&controlRef);
    HMSetControlHelpContent(controlRef,&helpContent);
  }
}

// ************************************************************************ helpTagsModeless

void  helpTagsModeless(DialogRef dialogRef)
{
  HMHelpContentRec helpContent;
  SInt16           a;
  static SInt16    itemNumber[7] = { 1,2,3,4,5,6,7 };
  ControlRef       controlRef;

  memset(&helpContent,0,sizeof(helpContent));

  HMSetTagDelay(500);
  HMSetHelpTagsDisplayed(true);

  helpContent.version = kMacHelpVersion;
  helpContent.tagSide = kHMOutsideTopCenterAligned;
  helpContent.content[kHMMinimumContentIndex].contentType = kHMStringResContent;
  helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmResID = 131;

  for(a = 1;a <= 7; a++)
  {
    helpContent.content[kHMMinimumContentIndex].u.tagStringRes.hmmIndex = a;
    GetDialogItemAsControl(dialogRef,itemNumber[a - 1],&controlRef);
    HMSetControlHelpContent(controlRef,&helpContent);
  }
}

// *******************************************************************************************
```

Demonstration Program DialogsAndAlerts Comments

When this program is run, the user should:

- Invoke alerts and dialogs by choosing items in the Demonstration menu, noting window update/activation/deactivation and menu enabling/disabling effects.
- On Mac OS 8/9, choose Show Balloons from the Help menu and pass the cursor over the various items in the dialogs, noting the information in the help balloons. Also note the updating of alerts and dialogs, and of the window, behind the help balloon when the balloon closes.
- On Mac OS X, pause the cursor over the various items in the dialogs, noting the information in the help tags.
- Note the effects on the menus when the various alerts and dialogs are the front window.
- Click anywhere outside the modal alert and modal dialog when they are the frontmost window, noting that the only response is the system alert sound.
- Note that, when the movable modal alert and movable modal dialog are displayed:
  - The program can be sent to the background by clicking outside the alert or dialog and the document window, or by bringing another application to the foreground.
  - The program can be brought to the foreground again by clicking inside the alert or dialog, or the document window.
- Note that, when the movable modal dialog and (on Mac OS X) the window-modal (sheet) dialog are displayed, the Edit menu and its Cut, Copy, and Paste items are enabled, given that the edit text items in these dialogs always have keyboard focus.
- Note that, when the modeless dialog is displayed:
  - It behaves like a normal document window when the user:
    - Clicks outside it when it is the frontmost window.
    - Clicks inside it when it is not the frontmost window.
  - It can be hidden by clicking in the close box/button or by selecting Close from the File menu.
  - A modal alert, movable modal alert, modal dialog or movable modal dialog can be invoked "on top of" the modeless dialog.
  - The Edit menu and its Cut, Copy, Paste, and Clear items are enabled so as to support text editing in the edit text item.
- Note that all alerts and dialogs respond correctly to Return and Enter key presses, and that the modal alert, modal dialog and movable modal dialog also respond correctly to escape key and Command-period presses.
- Note that, when an alert or dialog 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.
- In the modal dialog, click on the checkboxes to change their settings, noting that the new settings are remembered when the dialog is dismissed using the OK button, but not remembered when the dialog is dismissed using the Cancel button. Also, choose items in the two pop-up menus, noting that the chosen item is displayed in the document window's window header.
- In the movable modal dialog, click on the radio buttons to change their settings, noting that the new setting is remembered when the dialog is dismissed using the OK button, but not remembered when the dialog is dismissed using the Cancel button. In the case of the clock control and edit text item, 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 item, 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 item.

- In the modeless dialog and (on Mac OS X) the window-modal (sheet) dialog, 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 item. Also note that, when the Search button is clicked (or the Return or Enter keys are pressed) the text in the edit text item is displayed in the document window's window header.

- On Mac OS X, when a window-modal (sheet) alert or dialog is showing, minimise the window into the Dock and then expand the window from the dock.

In the 'DITL' resources for the modal and movable modal dialogs, 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 'dlgx' resources, note that all feature flags are set except for the kDialogFlagsHandlesMovableModal flag in 'dlgx' resources for the modal dialog and modeless dialog. Thus dialogs have a root control and embedding hierarchy.

Although, for demonstration purposes, this program creates modal alerts and dialogs, it is emphasised that Aqua Human Interface Guidelines require that, on Mac OS X, applications use modal alerts and dialogs only in exceptional purposes. On Mac OS X, the vast majority of alerts and dialogs should be application-modal or window-modal.

On Mac OS X, explanatory Help tags are available for the modal, movable modal, and modeless dialogs. The associated source code (in the source code file HelpTags.c) is not explained in these comments because the provision of Help tags is incidental to the demonstration. For information on creating help tags, see Chapter 25. Note that it is also possible to display help tags on Mac OS 8/9; however, it is considered by the author that their "look" is somewhat at odds with the Platinum appearance, and that help balloons remain the most appropriate option for Mac OS 8/9.

The CodeWarrior project for this program adds the CarbonFrameWorksLib stub library because certain functions are used that are only available on Mac OS X. (See "Carbon and Available APIs" at Chapter 25.) Those functions (CreateStandardAlert, RunStandardAlert, GetStandardAlertDefaultParams, CreateStandardSheet, ShowSheetWindow, HideSheetWindow) are only called when the program is run on Mac OS X.

DialogsAndAlerts.h

defines

Constants are established for dialog resource IDs and for the item numbers of certain items in the item lists associated with the dialogs. rAlertStrings and rSheetStrings represent the resource IDs of 'STR#' resources holding strings for the message and informative text (label and narrative text in Mac OS 8/9 parlance) for the modal alert, movable modal alert, and window-modal (sheet) alert.

The values represented by kSearchModeless and kSheetDialog will be stored as the reference constant in the window object in, respectively, the modeless dialog and window-modal (sheet) dialog objects. This enables the program to distinguish between these two dialogs.

The penultimate block establishes constants representing the character codes for the Return, Enter, escape, and period keys.

Finally, MAX_UINT32 is defined as the maximum possible unsigned long value. This value will be assigned to WaitNextEvent's sleep parameter at program launch.

DialogsAndAlerts.c

Global Variables

gEventFilterUPP will be assigned a universal procedure pointer to an application-defined event filter (callback) function. 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.)

The three variables after gDone will store the current control value of the checkboxes in the modal dialog. The next variable will store the item number of the currently selected radio button in the movable modal dialog.

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

main

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 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. (Note that pressing the Return key before the 10 seconds has elapsed will also dispose of the dialog.)

The call to NewModalFilterUPP creates a universal procedure pointer for the event filter (callback) function.

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 variable that will be used as WaitNextEvent's sleep parameter (gSleepTime) is initially set to the maximum unsigned long value. Note that the value assigned to gSleepTime will be changed at certain points in the program.

When a NULL event is returned by WaitNextEvent, if the program is running on Mac OS 8/9, doIdle is called.

doIdle

doIdle is invoked, on Mac OS 8/9 only, whenever WaitNextEvent returns a null event.

If the front window is the modeless dialog, 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 control, and the call causes the control definition function to call TextEdit to blink the insertion point caret. (This call is not necessary on Mac OS X because controls on Mac OS X have built-in timers.)

doEvents

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

Note that, at the autoKey case, the function doKeyDown is called only if the Command key is not down. This is to prevent the Command-key equivalents for cut, copy, and paste from repeating when the user presses and holds down those Command-key equivalents while editing text in the modeless dialog's edit text item.

In this program, autoKey events generated while the Command key is down are also discarded in the event filter (callback) function eventFilter (see below). This means that the behaviour of the edit text item in the movable modal dialog will replicate that of the edit text item in the modeless dialog.

(Many commercial and shareware programs (BBEdit excepted) do not discard autoKey events in these circumstances. The author considers that to be an oversight.)

doMouseDown

doMouseDown handles mouse-down events. Mouse-downs in the content region and in the close box/button are of significance to the demonstration. If a mouse-down occurred in a close box/button, if TrackGoAway returns true, and if the window is the modeless dialog, doHideModelessDialog is called. (In this demonstration, the modeless dialog, but not the document window, has a close box/button.)

doKeyDown

doKeyDown handles all key-down and auto-key events.

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 the modeless dialog or window-modal (sheet) dialog, or in the 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 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 the modeless dialog dialog or window-modal (sheet) dialog:

- If the key pressed was the Return or Enter key, GetDialogItemAsControl is called to get a reference to the single push button control in the modeless dialog (item 1 in the item list). The push button is then highlighted for eight ticks (this has an effect on Mac OS 8/9 only), and then unhighlighted before a function is called to extract the text from the edit text item and display it in the document window's window header. doKeyDown then returns because it is not intended that the edit text item 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 item. (The calls to HiliteMenu briefly highlight the Edit menu to indicated 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 dialogs with edit text items.)

  - If neither the X, C, nor V key was pressed, the 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.

  - doKeyDown returns so as to bypass the second call to DialogSelect.

  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.

- 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 item.

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, a function for updating the document window is called.

If the event is for the modeless dialog or window-modal (sheet) dialog, DialogSelect is called to handle the event. DialogSelect calls BeginUpdate, DrawDialog, and EndUpdate to redraw the dialog's content area. To restrict the redraw to the update region, an alternative is to call BeginUpdate, UpdateDialog, and EndUpdate. (Recall here that update regions BeginUpdate and EndUpdate are irrelevant on Mac OS X.)

Note that, on Mac OS X, the call to DialogSelect is really only necessary in the case of the window-modal (sheet) dialog.

doUpdateDocument

doUpdateDocument simply fills the content region of the document window with a colour, using a pixel patter ('ppat') resource and a call to FillCRect for that purpose, and then calls a function which draws a window header frame and the current contents of gCurrentString.

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 function for activating/deactivating the document window is called, otherwise the function for activating/deactivating the modeless dialog is called.

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 doDrawMessage draws a window header frame in the window, and the current contents of gMessageString, in either the activated or deactivated mode.

doActivateDialogs

doActivateDialogs performs window activation and deactivation for the modeless dialog and window-modal (sheet) dialog.

DialogSelect is called to handle the event. If the modeless or window-modal (sheet) dialog is becoming active, DialogSelect activates all controls and, on Mac OS 8/9, redraws the one-pixel-wide modeless dialog frame in the active mode. If the modeless dialog is going to the back, DialogSelect deactivates all controls and, on Mac OS 8/9, redraws the one-pixel-wide modeless dialog frame in the inactive mode.

In the remaining code, if either the modeless dialog or window-modal (sheet) dialog is becoming active, the menus are adjusted and the global variable used in the sleep parameter of the WaitNextEvent function is assigned the value returned by GetCaretTime (the cursor-blinking interval set by the user in the General Controls control panel (Mac OS 8/9) and System preferences (Mac OS X)). This is necessary to ensure that null events will always be generated, and thus doIdle and IdleControls will be called (necessary on Mac OS 8/9 only), at an interval short enough to ensure insertion point caret blinking at the proper rate.

If if either the modeless dialog or window-modal (sheet) dialog is being deactivated, the sleep parameter for the WaitNextEvent function is reset to the maximum unsigned long value.

(Recall that changing the value in gSleepTime is irrelevant on Mac OS X because the function doIdle will not be called when the program is running on Mac OS X.)

doAdjustMenus

Note that, if the program is running on Mac OS X, and a call to IsWindowCollapsed reveals that the document window has been minimised to the dock, the menu items which invoke the window-modal (sheet) alert and (sheet) dialog are disabled.

doMenuChoice

doMenuChoice handles menu choices. If the choice was the Close item in the File menu, and if the front window is the modeless dialog, a function which hides that modeless dialog is called. (In this program, because the document window does not have a close box/button, the Close item is only enabled when the modeless dialog is the front window.)

doEditMenu

doEditMenu first determines whether the front window is the modeless dialog or window-modal (sheet) dialog (both of which have an edit text item). If so, a reference to the dialog is obtained, following which Dialog Manager functions are called to cut, copy, paste, or clear text as appropriate. The Dialog Manager, in conjunction with TextEdit, performs these operations.

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.

Note that, when the Modal Alert item is chosen, doModalAlerts is called with false is passed in the function's parameter. Note also that, when the Movable Modal Alert item is chosen, doMovableModalAlertonX is called if the program is running on Mac OS X, otherwise doModalAlerts is called with true passed in the function's parameter. This latter reflects the fact that, on Mac OS X, the movable modal alert will be created by a function that is not available on Mac OS 8/9.

doExplicitlyDeactivateDocument

doExplicitlyDeactivateDocument is called at the beginning of those functions which create modal and movable modal alerts and dialogs.

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

doModalAlerts

doModalAlerts creates, displays, manages, and disposes of the modal alert and, on Mac OS 8/9, the movable modal alert.

The call to doExplicitlyDeactivateDocument deactivates the document window.

At the next nine lines, values are assigned to the fields of a standard alert parameter structure. In sequence: the alert is to be modal or movable modal depending on the value received in the formal parameter movable; a help button is to be displayed; the event filter (callback) function used is to be the application-defined event filter (callback) function pointed to by the universal procedure pointer gEventFilterUPP; the default title for the OK push button is to be used; a Cancel push button is required, and is to have the default title for the Cancel button; an Other push button is required, and is to have the title "Other"; the default push button is to be the first push button (which will thus have the default ring drawn around it (Mac OS 8/9) or be pulsing blue (Mac OS X) 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 is to be displayed in the alert position on the parent window screen. (With regard to the last field, the constant kWindowDefaultPosition equates to kWindowAlertPositionParentWindowScreen.)

The calls to GetIndString retrieve the specified strings from the specified 'STR#' resource. These are passed in the inError and inExplanation parameters in the following call to StandardAlert.

The call to StandardAlert creates and displays the alert (specifying a Stop alert in the first parameter), and handles all user interaction (by internally calling ModalDialog), including dismissing the alert when either the OK, Cancel, or Other button is hit. The item hit is returned in StandardAlert's outItemHit parameter.

In a real application, the appropriate action would be taken, based on which push button was hit, following the call to StandardAlert; however, in this demonstration, the identity of the push button is simply drawn in the document window.

doMovableModalAlertOnX

doMovableModalAlertOnX creates, displays, manages, and disposes of the movable modal alert when the program is run on Mac OS X. (The functions used in doMovableModalAlertOnX to create, display, and manage the alert are not available on Mac OS 8/9.)

The call to doExplicitlyDeactivateDocument deactivates the document window.

The call to GetStandardAlertDefaultParams initialises a standard CFString alert parameter structure with default values. (The defaults are: not movable; no Help button; no Cancel button; no Other button; alert position on parent window screen.) The next four lines modify the defaults by specifying that the alert is to be movable modal, and is to have a Help, Cancel, and Other, push button.

The two calls to GetIndString retrieve the specified strings from the specified 'STR#' resource, which are then converted to CFStrings before being passed in the error and explanation fields in the following call to CreateStandardAlert. CreateStandardAlert creates the alert.

The call to RunStandardAlert displays the alert and runs the alert using a ModalDialog loop. When a push button is clicked, its item number is returned in the outItemHit parameter

In a real application, the appropriate action would be taken, based on which push button was hit, following the call to RunStandardAlert; however, in this demonstration, the identity of the push button is simply drawn in the document window.

doModalDialog

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

The call to doExplicitlyDeactivateDocument deactivates the document window.

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

The call to SetDialogDefaultItem tells the Dialog Manager which is the default push button item and aliases the Return and Enter keys to 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.

The call to NewMenu creates a new empty menu for a font menu. A reference to this menu is passed in the call to CreateStandardFontMenu, which creates a non-hierarchical font menu containing the names of all resident fonts. GetDialogItemAsControl gets a reference to the Font pop-up menu button control, facilitating the calls to SetControlMinimum, SetControlMaximum, and SetControlData. The latter sets the menu to be used by the pop-up menu button control.

With the modal dialog 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 reference 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 reference to the control and GetControlValue is called to get the menu item number of the menu item chosen, following which an function is called to extract the menu item text and display it in the window header.

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

doMovableModalDialog

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

The call to doExplicitlyDeactivateDocument deactivates the document window.

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

The call to SetDialogDefaultItem tells the Dialog Manager which is the default push button item and aliases the Return and Enter keys to 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 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 item.

The first call to GetDialogItemAsControl gets a reference 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 reference 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 oldBrushTupe. (This may be required later.)

With the movable modal dialog 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 reference 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 (callback) function. Note also that all user interaction relating to the clock control and edit text item 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 item would possibly be retrieved at this point if the user hit the OK push button.)

doCreateOrShowModelessDialog

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

Accordingly, the first line determines whether the modeless dialog is already open. If it is not: the call to GetNewDialog creates the modeless dialog; the call to SetWRefCon assigns the reference constant kSearchModeless to the dialog object's window object so as to differentiate this dialog from the window-modal (sheet) dialog; a call to SetDialogDefaultItem causes the push button to be drawn with the default ring (Mac OS 8/9) or in pulsing blue (Mac OS X); a call to SetDialogItemText assigns some initial text to the edit text item; a call to SelectDialogItemText selects the text in the edit text item. (Note that, if the edit text item 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 function doIdle (Mac OS 8/9).)

If, on the other hand, the modeless dialog has already been opened, the call to ShowWindow displays the dialog 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.

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

If the event occurred in the modeless dialog or the window-modal (sheet) dialog (both of which contain an edit text item), 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 item. If DialogSelect returns true, and if the item hit was the OK push button, a 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. It simply demonstrates retrieval of the text in an edit text item in a dialog.

The call to GetDialogItemAsControl gets a reference to the edit text control, and the call to GetDialogItemText copies the text in the edit text control to the global variable gCurrentString. The following lines cause that text to be drawn in the window header in the document window.

doHideModelessDialog

doHideModelessDialog hides a modeless dialog. The call to HideWindow makes the dialog invisible. The sleep parameter for the WaitNextEvent function is reset to the maximum possible long value (relevant only on Mac OS 8/9), because insertion point caret blinking is not required while the Search dialog is hidden.

eventFilter

eventFilter is the application-defined event filter (callback) function which, in conjunction with ModalDialog, handles events in the modal alerts, movable modal alert on OS 8/9, modal dialog and movable modal dialog.

If the program is running on Mac OS 8/9, if the event is an update event, and if that event is not for the dialog or alert 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. The call to the window updating function is not necessary on Mac OS X.

If the event is an autoKey event and the Command key is down, the event is, in effect, discarded. This means that, if the user is working within the edit text item in the movable modal dialog and presses and holds the Command key equivalents for Cut, Copy or Paste, repeating cut, copy and paste actions will be defeated. That is, pressing and holding the Command key equivalent will only result in a single cut, copy, or paste action. (See also doEvents, above.)

If the event is neither an update event nor an autoKey event with the Command key down, the current graphics port is saved and then set to that of the alert or dialog. The event is then passed to the standard event filter (callback) function for handling. If the standard event filter (callback) function handles the event, it will return true and, in the itemHit parameter, the number of the item that it handled. ModalDialog 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 (callback) function is used by all but the movable modal dialog. 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 is the only dialog which requires this tracking (because it contains an edit text item.)

doPopupMenuChoice, doPlaySound, doDrawMessage, and doCopyPString

doPopupMenuChoice, doPlaySound, doDrawMessage, and doCopyPString are incidental to the demonstration. All perform the same duties as the similarly-named functions in the demonstration program Controls1 (Chapter 7). doDrawMessage is used in this program to prove the explicit deactivation of the document window's content area when alerts and dialogs other than the modeless dialog are invoked.

Sheets.c

Global Variables

gSheetDialogWindowRe will be assigned the window reference of the window-modal (sheet) dialog.

doSheetAlert

doSheetAlert creates, displays and handles a window-modal (sheet) alert.

The call to GetStandardAlertDefaultParams initialises a standard CFString alert parameter structure with default values.

The two calls to GetIndString retrieve the specified strings from the specified 'STR#' resource, which are then converted to CFStrings before being passed in the error and explanation fields in the following call to CreateStandardSheet, which creates the alert. The call to ShowSheetWindow displays the sheet. When the user clicks the OK button, the sheet is dismissed.

doSheetDialog

doSheetAlert creates and displays a window-modal (sheet) dialog.

The call to GetNewDialog creates the dialog from the specified resource. The call to SetWRefCon assigns a value as the dialog window's reference constant. This is used elsewhere to differentiate the window-modal (sheet) dialog from the modeless dialog.

The call to SetDialogDefaultItem causes the push button to be drawn with the default ring (Mac OS 8/9) or in pulsing blue (Mac OS X). The call to SetDialogItemText assigns some initial text to the edit text item and the call to SelectDialogItemText selects that text.

The next line assigns a reference to the dialog's window to the global variable gSheetDialogWindowRef. This is used in the function doButtonHitInSheetDialog.

The call to ShowSheetWindow displays the sheet.

doButtonHitInSheetDialog

doButtonHitInSearchModeless is called from doKeyDown and doInContent to further process, to completion, a hit on the OK button in the window-modal (sheet) dialog. In addition to retrieving the text in an edit text item, it hides and disposes of the dialog.