DIALOG MANAGER

Jean-Charles Mourey

6/12/86	Beginning Dialog Manager
6/30/86	First Draft
7/11/86	Second Draft (new calls using templates, alerts, item change on the fly,...)
7/14/86	Third Draft - First official release
8/5/86	Fourth Draft - Second official release (new parameter to NewDItem,...)
8/12/86	Fifth Draft - modifications and enhancements + new calls + new section.

ABOUT THIS MANUAL

This chapter describes the Dialog Manager, the tool that allows you to implement dialog boxes and the alert mechanism, two means of communication between the application and the end user.

You should already be familiar with:

- the basic concepts and structures behind QuickDraw, particularly rectangles, grafPorts, and pictures.
- the Event Manager, the Window Manager, and the Control Manager.
- LineEdit, to understand editing text in dialog boxes.

Note: This manual intends, wherever possible, to describe the final version of the Dialog Manager. Several features might not be implemented yet. When so, a *** message is usually added to signal this fact. Also, some errors or inconsistencies may remain in this draft. Please read this manual and report any problem to:

Jean-Charles Mourey (x6624)
MS 22-X
Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014

AppleLink address: Mourey1

DIFFERENCES WITH THE MACINTOSH

The main difference between the Macintosh Dialog Manager and the Cortland Dialog Manager is that there is no Resource Manager on the Cortland. Since the Mac Dialog Manager is almost entirely written to be used with a Resource Manager, the Cortland Dialog Manager had to provide some additionnal facilities to the developer to make it useful and powerful at the same time.

The Cortland Dialog Manager provides 58 procedures and functions, allowing the programmer to create dialogs and items in various ways, to handle dialogs and alerts, to modify the look and behavior of the items, and to make some general modifications.

Dialogs

It has been decided that a dialog cannot be modal and modeless at the same time. There is different calls for creating modal dialogs and modeless dialogs. This allows to create a modal dialog more easily and the Dialog Manager does some additional checking like preventing the use of a modal dialog as a modeless dialog and vice-versa.

It is still possible to create a modal and a modeless dialog with the same items by using the same item templates for the two dialogs.

There is two ways to create a modal dialog: NewModalDialog and GetNewModalDialog. The second one is more easy to use since it gets all the necessary information from a template.

Items

The Mac Dialog Manager does not allow the creation or removal of items on the fly, since they are all defined in resources.

On the Cortland, items may be created at any time, using stack paramaters or templates.

The following types are supported:

Buttons, Check Boxes,

Radio buttons,

Set of radio buttons (not implemented on the Mac),

Scroll bars with a user procedure that allows you to properly handle scroll bar dragging and change some other items in the dialog while the user is scrolling (not implemented on the Mac),

Custom Controls,

Static Text (but you have to place the carriage returns yourself),

Long Static Text (with up to 32767 characters), Editable text (with only one line of text, but with automatic Cut/Copy/Paste),

Icon of any size (as long as their width is a multiple of 8),

User item.

The length of an editable text may be limited to a specific value (useful for example to limit the length of Prodos file names to 15 characters).

It is possible to specify the initial value of an item at the time of creation.

Icons and user items (and also StatText) may have a "value" that the application can use for its own purpose.

Items can be added, removed, hidden or shown on the fly.

The scroll bar is a standard dialog item an is completely handled by the Dialog Manager (unlike the Macintosh) by using your dialog scrollbar action procedure.

There is a lot of short cuts and features implemented for editing a line of editable text. Most of them are short cuts implemented by LineEdit and another one is the Apple-X/C/V feature which allow to cut/copy/paste inside or between editable items and between dialogs.

When using modal dialogs and alerts, there is a standard filter that implements the automatic Cut/Copy/Paste feature and the Return as equivalent to the default button. It is possible to use a custom filter without losing the benefits of the default filter (On the Macintosh, a custom filter had to reimplement the standard features).

It is possible to go through the list of items with GetFirstItem and GetNextItem and make whatever action you want on each item of the list.

The concept of default button is automatically handled by the Dialog Manager. A default button is visually recognized by its bold outline and is functionnaly equivalent to the Return key. Initially the default button is the item with an ID 1, and it can be changed on the fly with special procedures.

In the Macintosh, a dialog maintains two lists, an item list and a control list containing all the items that are controls. It means that controls in dialog are referenced two times, one time in the item list and one time in the control list.
In the Cortland, all the items are implemented as controls. The regular controls defined by the Control Manager are just standard controls (except for the Scroll Bar which definition is extended to accept user action procedure). The other items are implemented as custom controls.
This is why there is a procedure called GetControlItem that gives you the handle to the control defining the corresponding item. Although it is preferrable to use Dialog Manager calls, you can virtually make any Control Manager call on this control. GetControlItem would be used to make certain very specific Control Manager calls (for scroll bars for example). It is highly recommended to use it as rarely as possible.

Everything else is working like on the Macintosh.

ABOUT THE DIALOG MANAGER

The Dialog Manager is a tool for handling dialogs and alerts in a way that's consistent with the Apple User Interface Guidelines.

A dialog box appears on the screen when an application needs more information to carry out a command. As shown in Figure 1, it typically resembles a form on which the user checks boxes and fills in blanks.

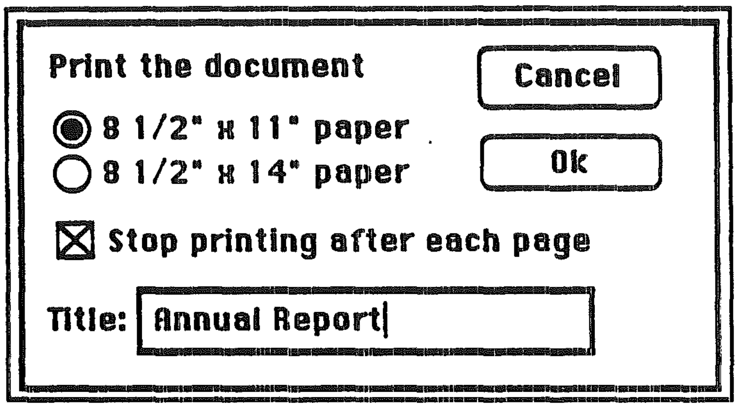

Figure 1. A Typical Dialog Box

By convention, a dialog box comes up slightly below the menu bar, is somewhat narrower than the screen, and is centered between the left and right edges of the screen. It may contain any or all of the following:

- informative or instructional text
- rectangles in which text may be entered (initially blank or containing default text that can be edited)
- controls of any kind
- graphics (icons or QuickDraw pictures)
- anything else, as defined by the application.

The user provides the necessary information in the dialog box, such as by entering text or clicking a check box. There's usually a button labeled "OK" to tell the application to accept the information provided and perform the command, and a button labeled "Cancel" to cancel the command as though it had never been given (retracting all actions since its invocation). Some dialog boxes may use a more descriptive word than "OK"; for simplicity, this manual will still refer to the button as the "OK button". There may even be more than one button that will perform the command, each in a different way.

Most dialog boxes require the user to respond before doing anything else. Clicking a button to perform or cancel the command makes the box go away; clicking outside the dialog box only causes a beep from the Cortland's speaker. This type is called a modal dialog box because it puts the user in the state or "mode" of being able to work only inside the dialog box. A modal dialog box usually has the same general appearance as shown in Figure 1 above. One of the buttons in the dialog box may be outlined boldly. Pressing the Return key has the same effect as clicking the outlined button or, if none, the OK button; the particular button whose effect occurs is called the dialog's default button and is the preferred ("safest") button to use in the current situation. If there's no boldly outlined or OK button, pressing Return will by convention have no effect.

Other dialog boxes do not require the user to respond before doing anything else; these are called modeless dialog boxes (see Figure 2). The user can, for example, do work in document windows on the desktop before clicking a button in the dialog box, and modeless dialog boxes can be set up to respond to the standard editing commands in the Edit menu. Clicking a button in a modeless dialog box will not make the box go away: The box will stay around so that the user can perform the command again. A Cancel button, if present, will simply stop the action currently being performed by the command; this would be useful for long printing or searching operations, for example..

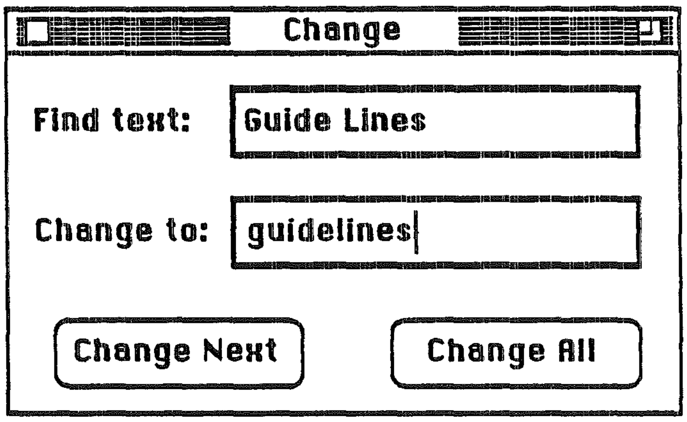

Figure 2. A Modeless Dialog Box

As shown in Figure 2, a modeless dialog box looks like a document window. It can be moved, made inactive and active again, or closed like any document window. When you're done with the command and want the box to go away, you can click its close box or choose Close from the File menu when it's the active window.

Dialog boxes may in fact require no response at all. For example, while an application is performing a time-consuming process, it can display a dialog box that contains only a message telling what it's doing; then, when the process is complete, it can simply remove the dialog box.

The alert mechanism provides applications with a means of reporting errors or giving warnings. An alert box is similar to a modal dialog box, but it appears only when something has gone wrong or must be brought to the user's attention. Its conventional placement is slightly farther below the menu bar than a dialog box. To assist the user who isn't sure how to proceed when an alert box appears, the preferred button to use in the current situation is outlined boldly so it stands out from the other buttons in the alert box (see Figure 3). The outlined button is also the alert's default button; if the user presses the Return key, the effect is the same as clicking this button.

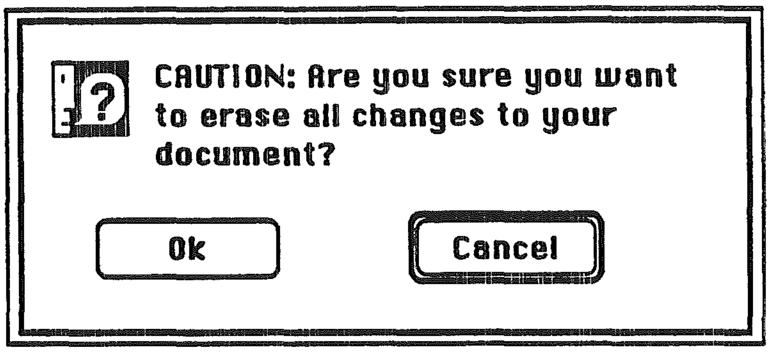
CAUTION: Are you sure you want to erase all changes to your document?
Ok
Cancel

Figure 3. A Typical Alert Box

There are four standard kinds of alerts—Stop, Note, Caution and Talk—each indicated by a particular icon in the top left corner of the alert box. Figure 3 illustrates a Caution alert. The icons identifying Stop, Note and Talk alerts are similar; instead of a question mark, they show an exclamation point, an asterisk and a talking face, respectively. Other alerts can have anything in the the top left corner, including blank space if desired.

The alert mechanism also provides another type of signal: Sound from the Cortland's speaker. The application can base its response on the number of consecutive times an alert occurs; the first time, it might simply beep, and thereafter it may present an alert box. The sound isn't limited to a single beep but may be any sequence of tones, and may occur either alone or along with an alert box. As an error is repeated, there can also be a change in which button is the default button (perhaps from OK to Cancel). You can specify different responses for up to four occurrences of the same alert.

With Dialog Manager routines, you can create dialog boxes or invoke alerts.

DIALOG AND ALERT WINDOWS

A dialog box appears in a dialog window. When you call a Dialog Manager routine to create a dialog, you supply the same kind of information as when you create a window with a Window Manager routine. For example, you call [Get]NewModalDialog or [Get]NewModelessDialog, which determines how the window looks and behaves, you supply a rectangle that becomes the portRect of the window's grafPort, and you specify whether the window is visible or invisible. For modeless dialogs, you also specify the window's plane (which, by convention, should initially be the frontmost). The dialog window is created as specified.

You can manipulate a dialog window just like any other window with Window Manager or QuickDraw routines, showing it, hiding it, moving it, changing its size or plane, or whatever— all, of course, in conformance with the Apple User Interface Guidelines. The Dialog Manager observes the clipping region of the dialog window's grafPort, so if you want clipping to occur, you can set this region with a QuickDraw routine.

Similarly, an alert box appears in an alert window. You don't have the same flexibility in defining and manipulating an alert window, however. The Dialog Manager chooses the window definition procedure, so that all alert windows will have the standard appearance and behavior. The size and location of the box are supplied as part of the definition of the alert and are not easily changed. You don't specify the alert window's plane; it always comes up in front of all other windows. Since an alert box requires the user to respond before doing anything else, and the response makes the box go away, the application doesn't do any manipulation of the alert window.

Figure 4 illustrates a document window, dialog window, and alert window, all overlapping on the desktop.

Figure 4. Dialog and Alert Windows

ITEM LISTS

To create a dialog or an alert, the Dialog Manager needs some information in order to create the dialog or alert window. The Dialog Manager also needs to know what items the dialog or alert box contains.

This section discusses the contents of an item list once it is created and the Dialog Manager has set it up as necessary to be able to work with it.

An item list contains the following information for each item:

- An ID number uniquely identifying the item in the dialog. All subsequent Dialog Manager calls referring to that item will be made using the ID number.
- The type of item. This includes not only whether the item is a standard control, text, or whatever, but also whether the Dialog Manager should return to the application when the item is clicked.
- An item descriptor like a title for a control, a procedure pointer for a user-defined item or text for an editable or non-editable text item.
- A display rectangle, which determines the location of the item within the dialog or alert box.
- The initial value of a standard control, the word length of a LongStatText item, the maximum string length of an EditLine item, or any value you want for a UserItem.
- A flag determining whether the item should be created visible or invisible and including item-specific information, like the family number of a radio button, or whether a scroll bar is horizontal or vertical.
- A color table allowing to change the standard colors used to draw items in a dialog. It may be useful for example to draw red-colored text (Custom color tables are used only for standard controls or controls you define yourself).

There's several Dialog Manager procedures that, given a pointer to a dialog port and an item ID, sets or returns that item's text, type, display rectangle, flags, value and color.

Item Types

The item type is specified by a predefined constant or combination of constants, as listed below. Figure 5 illustrates some of these item types.

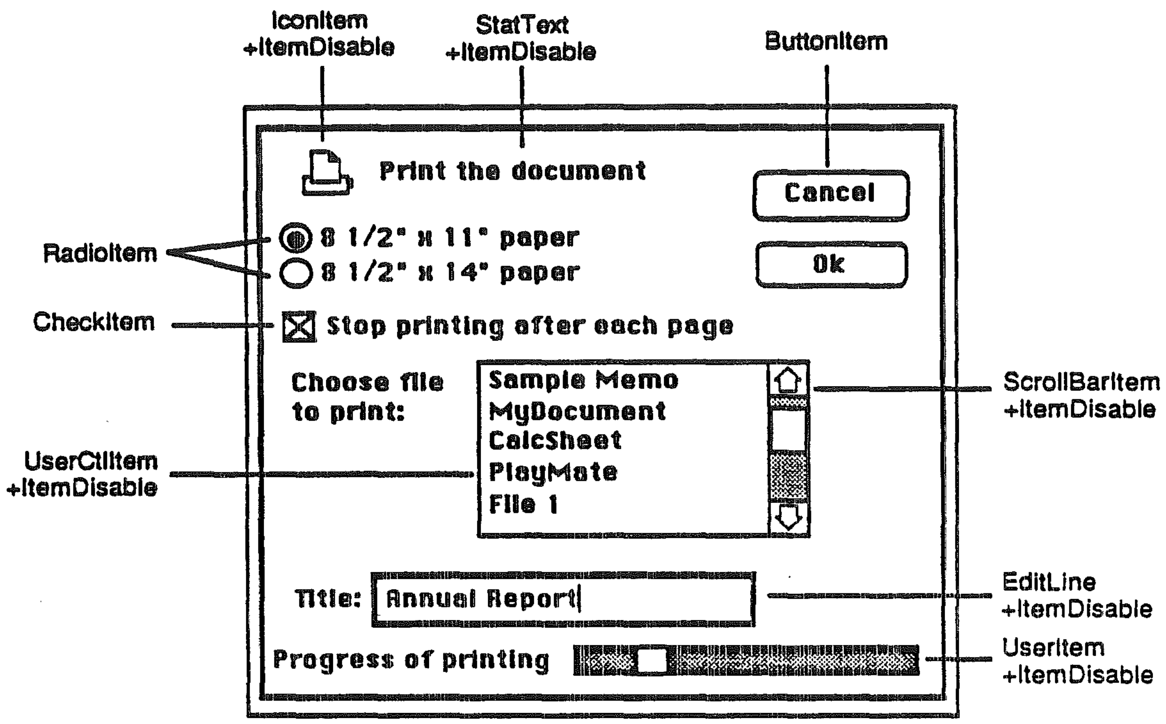

Figure 5. Item Types

Item type	Meaning
ButtonItem	A standard button control.
CheckItem	A standard check box control.
RadioItem	A standard radio button control.
ScrollBarItem	A special kind of scroll bar for dialogs.
UserCtlItem	A control defined by the application.
StatText	Static text; text that cannot be edited (Several lines allowed).
EditLine	(Dialogs only) Text that can be edited; the Dialog Manager accepts text typed by the user and allows editing (One line only).

IconItem	An icon.
PicItem	A QuickDraw picture.
UserItem	(Dialogs only) An application-defined item, such as a picture whose appearance changes.
ItemDisable+<any of the above>	The item is disabled (the Dialog Manager doesn't report events involving this item).

The text of an EditLine item may initially be either default text or empty. Text entry and editing is handled in the conventional way, as in LineEdit—in fact, most of the time, the Dialog Manager calls LineEdit to handle it:

- Clicking in the item displays a blinking vertical bar, indicating an insertion point where text may be entered.
- Dragging over text in the item selects that text, double-clicking selects a word, and triple-clicking selects the line; the selection is highlighted and then replaced by what the user types.
- Double-clicking followed by dragging extends or shortens the selection by one word at a time.
- The left and right arrow keys move the insertion point left or right by one character.
- Using the left and right arrow keys while holding down the Apple key moves the insertion point to the beginning or the end of the line.
- Using the left and right arrow keys while holding down the Option key moves the insertion point left or right by one word at a time.
- Clicking or using the arrows with any modifiers while holding down the Shift key extends or shortens the current selection.
- The Backspace key deletes the current selection or the character preceding the insertion point.
- Control-F deletes the character following the insertion point or the current selection.
- Control-Y deletes from the insertion point to the end of the line, or the current selection.
- Control-X deletes the whole line or the current selection.
- The Apple-X, Apple-C, Apple-V commands respectively Cut, Copy and Paste the current selection in the active EditLine item, allowing to copy and paste text between different EditLine items (Note: the cut/copy/paste mechanism preserves the space between words).

The Tab key advances to the next EditLine item in the item list, wrapping around to the first if there aren't any more. In an alert box or a modal dialog box (regardless of whether it contains an EditLine item), the Return key has the same effect as clicking the default button; for alerts, the default button is identified in the alert template, whereas for modal dialogs it's the item in the item list whose ID number is 1 (unless specified otherwise).

If ItemDisable is specified for an item, the Dialog Manager doesn't let the application know about events involving that item. For example, you may not have to be informed every time the user types a character or clicks in an EditLine item, but may only need to look at the text when the OK button is clicked. In this case, the EditLine item would be disabled. Standard buttons and check boxes should always be enabled, so your application will know when they've been clicked.

Warning: Don't confuse disabling a control with making one "inactive" with the Control Manager procedure HiliteControl: When you want a control not to respond at all to being clicked, you make it inactive. An inactive control is highlighted to show that it's inactive, while disabling a control doesn't affect its appearance.

Item Descriptor and Value

The item descriptor and value contain the following information for the various types of items:

Item type	**Item Descriptor**	**ItemValue**
ButtonItem	A pointer to the title string	init value of the control
CheckItem	A pointer to the title string	init value of the control
RadioItem	A pointer to the title string	init value of the control
ScrollBarItem	A Dialog ScrollBar Action procedure pointer	0 or default value if ItemDescr=0
UserCtlItem	A control definition procedure pointer	init value of the control
StatText	A pointer to the static string	application-use
LongStatText	A pointer to the beginning of the text	length of the text (0 to 32767)
EditLine	A pointer to the default string	maximum allowed length (0 to 255)
IconItem	A handle to the icon	application-use
PicItem	A picture handle	application-use
UserItem	An item definition procedure pointer	application-use

Note: Whenever "application-use" is specified under "Item Value", it means that the value parameter is not accessed by the Dialog Manager and can be used by the application for its own purpose (use GetItemValue and SetItemValue to change this field). For example, the application might want to store there the position of the indicator of the useritem in figure 5. Note that SetItemValue redraws the item to display its new value.

The procedure for a userItem draws the item; for example, if the item is a clock, it will draw the clock with the current time displayed. When this procedure is called, the current port will have been set by the Dialog Manager to the dialog window's grafPort. The procedure must have two parameters, a dialog pointer and an item ID. For example, this is how it would be declared if it were named MyItem:

MyItem

```
  input:  theDialog:LONG        pointer to the dialog's grafport
          itemID:WORD           ID of item to draw

  output: none
```

TheDialog is a pointer to the dialog window; in case the procedure draws in more than one dialog window, this parameter tells it which one to draw in.
ItemID is the item ID; in case the procedure draws more than one item, this parameter tells it which one to draw.

Display Rectangle

Each item in the item list is displayed within its display rectangle:

- For standard controls, scroll bars and user controls, the display rectangle becomes the control's enclosing rectangle.
- For an EditLine item, it becomes LineEdit's view rectangle. The text is clipped if there's more than will fit in the rectangle. In addition, the Dialog Manager uses the QuickDraw procedure FrameRect to draw a rectangle outside the display rectangle.
- StatText and LongStatText items are displayed in exactly the same way as EditLine items, except that a rectangle isn't drawn outside the display rectangle and it is possible to have more than one line of text by inserting carriage return characters in the text.
- Icons and QuickDraw pictures are scaled or clipped to fit the display rectangle. For pictures, the Dialog Manager calls the QuickDraw procedure DrawPicture and passes it the display rectangle.
- If the procedure for a userItem draws outside the item's display rectangle, the drawing is clipped to the display rectangle.

Note: Clicking anywhere within the display rectangle is considered a click in that item. If display rectangles overlap, a click in the overlapping area is considered a click in whichever item comes first in the item list.

By setting the invisible flag, you can make the item invisible. This might be useful, for example, if your application needs to display a number of dialog boxes that are similar except that one item is missing or different in some of them. You can use a single dialog box in which the item or items that aren't currently relevant are invisible. To remove an item or make one reappear, you just make it visible or invisible. Note the following, however:

- Instead of making an item invisible and visible, you can use the RemoveItem procedure which completely removes the item from the item list.
- The rectangle for a statText item must always be at least as wide as the first character of the text; a good rule of thumb is to make it at least 20 pixels wide.
- To change text in a statText item, it's easier to use the Dialog Manager procedure ParamText (as described later in the "Dialog Manager Routines" section).

Item ID

Each item in an item list is identified by an item ID, a unique number in the list allowing you to further reference this item. By convention, the OK button in an alert's item list should have an ID of 1 and the Cancel button should have an ID of 2. The Dialog Manager provides predefined constants equal to the item ID for OK and Cancel:

```
ok          equ   1
cancel      equ   2
```

In a modal dialog's item list, the item whose ID is 1 is assumed to be the dialog's default button (unless specified otherwise); if the user presses the Return key, the Dialog Manager normally returns the ID of the default button, just as when that item is actually clicked.

In conformance to the Apple User Interface Guidelines, the Dialog Manager automatically boldly-outline the default button, unless there is no default button (no button item with ID 1).

Note: If you don't want any default button, you should not create any item with an ID 1.

Item Flag

The Item Flag parameter contains most of the time the same as the Ctrl Flag value that you would pass to NewControl to create a control. It may contain the family number of a radio button, or whether a scroll bar is vertical or horizontal. For more details, see the Control Manager ERS.

Note: You should not use itemFlag to boldly-outline a button, since the Dialog Manager handles the concept of default button for you.

DIALOG RECORDS

To create a dialog, you pass information to the Dialog Manager in parameters or in a template; the Dialog Manager incorporates the information into a dialog record. The dialog record contains the window record for the dialog window, a handle to the dialog's item list, and some additional fields. The Dialog Manager creates the dialog window by calling the Window Manager function NewWindow and then setting the dialog type in the dialog record to indicate whether it is a modal or a modeless dialog. The routine that creates the dialog returns a pointer to the dialog port, which you use thereafter to refer to the dialog in Dialog Manager routines or even in Window Manager or QuickDraw routines (see "Dialog Pointers" below). The Dialog Manager provides routines for handling events in the dialog window and disposing of the dialog when you're done.

You can do all the necessary operations on a dialog without accessing the fields of the dialog record directly.

Dialog Pointers

A DialogPtr is a pointer to the dialog's port, very much as a WindowPtr. Don't assume that it is also a pointer to the dialog record or even the window record, because it's NOT!

Accessing the Dialog Record and the Item Records

The structure of a Dialog Record is private.

To get or change information about a dialog, you pass the dialog pointer to a Dialog Manager routine. You'll never access information directly in the record.

To get or change information about an item in a dialog, you pass the dialog pointer and the item ID to a Dialog Manager routine. You'll never access information directly through the handle to the item, except for some rare cases.

ALERTS

When you call a Dialog Manager routine to invoke an alert, you pass it a pointer to the alert template, which contains the following:

- An alert ID used by the Dialog Manager to deal with stages between the different alerts.
- A rectangle, given in global coordinates, which determines the alert window's size and location. It becomes the portRect of the window's grafPort. To allow for the menu bar and the border around the portRect, the top coordinate of the rectangle should be at least 25 points below the top of the screen.
- Information about exactly what should happen at each stage of the alert.
- A list of items.

Every alert has four stages, corresponding to consecutive occurrences of the alert: The first three stages correspond to the first three occurrences, while the fourth stage includes the fourth occurrence and any beyond the fourth. (The Dialog Manager compares the current alert's ID to the last alert's ID to determine whether it's the same alert.) The actions for each stage are specified by the following three pieces of information:

- which is the default button—the OK button (or, if none, a button that will perform the command) or the Cancel button. For an alert, the Ok button shoud have an ID 1 and the Cancel button an ID 2.
- whether the alert box is to be drawn
- which of four sounds should be emitted at this stage of the alert

The alert sounds are determined by a sound procedure that emits one of up to four tones or sequences of tones. The sound procedure has one parameter, an integer from 0 to 3; it can emit any sound for each of these numbers, which identify the sounds in the alert template. For example, you might declare a sound procedure named MySound as follows:

```
MySound

  input:  SoundNo:WORD          number of the sound

  output: none
```

If you don't write your own sound procedure, the Dialog Manager uses the standard one: Sound number 0 represents no sound and sound numbers 1 through 3 represent the corresponding number of short beeps, each of the same pitch and duration. The volume of each beep depends on the current speaker volume setting, which the user can adjust with the Control Panel desk accessory.

For example, if the second stage of an alert is to cause a beep and no alert box, you can just specify the following for that stage in the alert template: Don't draw the alert box, and use sound number one. If instead you want, say, two successive beeps of different pitch, you need to write a procedure that will emit that sound for a particular sound number, and specify that number in the alert template. The Cortland Miscellaneous Tools FWEntry procedure allows to call from a 16-bit environment the Apple // firmware which has routines for emitting sound (The standard sound procedure calls the BELL routine at $FBDD); for more complex sounds, you can use the Sound Tools.

Note: When the Dialog Manager detects a click outside an alert box or a modal dialog box, it emits sound number 1; thus, for consistency with the Apple User Interface Guidelines, sound number 1 should always be a single beep.

Internally, alerts are treated as special modal dialogs. The alert routine creates the alert window by calling NewModalDialog and every item with GetNewDItem. The Dialog Manager works with the dialog created by NewModalDialog, just as when it operates on a dialog window, but it disposes of the dialog before returning to the application. Normally your application won't change the dialog record for an alert; however, there is a way that this can happen: for any alert, you can specify a procedure that will be executed repeatedly during the alert, and this procedure may access the dialog. For details, see the alert routines under "Invoking Alerts" in the "Dialog Manager Routines" section.

USING THE DIALOG MANAGER

Before using the Dialog Manager, you must initialize the Memory Manager, QuickDraw, the Event Manager, the Window Manager, the Control Manager, and LineEdit, in that order. The first Dialog Manager routine to call is DialogStartup, which initializes the Dialog Manager. If you want the font in your dialog and alert windows to be other than the system font, call SetDAFont to change the font.

Where appropriate in your program, call NewModalDialog, NewModelessDialog or GetNewModalDialog to create any dialogs you need. Then call NewDItem or GetNewDItem for each new item you want to add to the dialog. When you no longer need a dialog, you'll usually call CloseDialog.

In most cases, you probably won't have to make any changes to the dialogs from the way they're defined at their creation. However, if you should want to modify an item in a dialog, you can call one of the GetItemXXX calls to get information about the item and SetItemXXX to change it. In some cases it may be appropriate to call some other routine to change the item; for example, to move a control in a dialog, you would get its handle from GetControlItem and then call the appropriate Control Manager routine. There are also two procedures specifically for accessing or setting the content of a text item in a dialog box: GetIText and SetIText.

To handle events in a modal dialog, just call the ModalDialog procedure after putting up the dialog box. If your application includes any modeless dialog boxes, you'll pass events to IsDialogEvent to learn whether they need to be handled as part of a dialog, and then usually call DialogSelect if so. Before calling DialogSelect, however, you should check whether the user has given the keyboard equivalent of a command, and you may want to check for other special cases, depending on your application. You can support the use of the standard editing commands in a modeless dialog's editText items with DlgCut, DlgCopy, DlgPaste, and DlgDelete.

A dialog box that contains EditLine items normally comes up with the insertion point in the first such item in its item list. You may instead want to bring up a dialog box with text selected in an EditLine item, or to cause an insertion point or text selection to reappear after the user has made an error in entering text. For example, the user who accidentally types nonnumeric input when a number is required can be given the opportunity to type the entry again. The SelIText procedure makes this possible.

For alerts, if you want other sounds besides the standard ones (up to three short beeps), write your own sound procedure and call ErrorSound to make it the current sound procedure. To invoke a particular alert, call one of the alert routines: StopAlert, NoteAlert, CautionAlert, or TalkAlert for one of the standard kinds of alert, or Alert for an alert defined to have something other than a standard icon (or nothing at all) in its top left corner.

Finally, you can substitute text in StatText items with text that you specify in the ParamText procedure. This means, for example, that a document name supplied by the user can appear in an error message.

DIALOG MANAGER ROUTINES

Initialization and ShutDown

```
DialogBootInit                Call # 1

  input:  none

  output: none
```

DialogBootInit is called at initialization time. It does nothing.

```
DialogStartup                 Call # 2

  input:  ProgramID:WORD      ID to use with the Memory Manager

  output: none
```

Call DialogStartup once before all other Dialog Manager routines, to initialize the Dialog Manager. DialogStartup does the following initialization:

- It installs the standard sound procedure.
- It passes empty strings to ParamText.
- It sets the dialog font to the System font.
- It sets the alert stage to 1.

Note: The Dialog Manager shares its zero page with the Control Manager, so it does not need a special zero page. Note that the Control Manager must be present in order for the Dialog Manager to run, even if you do not have any standard control items in your dialogs.

```
ErrorSound                    Call # 7

  input:  soundProc:LONG      pointer to a sound procedure

  output: none
```

ErrorSound sets the sound procedure for alerts to the procedure pointed to by soundProc; if you don't call ErrorSound, the Dialog Manager uses the standard sound procedure. (For details, see the "Alerts" section.) If you pass NIL for soundProc, the standard procedure will be used instead.

Note: The sound procedure is also called by ModalDialog with a Sound Number of 1, when the user clicks outside the dialog.

SetDAFont Call # 8

```
  input:  FontHandle:LONG        handle to the new font

  output: none
```

For subsequently created dialogs and alerts, SetDAFont causes the font of the dialog or alert window's grafPort to be set to the font having the specified font number. If you don't call this procedure, the system font is used. SetDAFont affects statText, EditLine items and standard controls.

DialogShutDown Call # 3

```
  input:  none

  output: none
```

DialogShutDown shuts down the Dialog Manager and frees up any memory allocated by the Dialog Manager.

DialogVersion Call # 4

```
  input:  none

  output: dVersion:WORD - Dialog Manager's version number.
```

DialogReset Call # 5

```
  input:  none

  output: none
```

DialogReset resets the dialog font to the system font, clears the ParamText strings, reset the sound procedure to the standard sound procedure and reset the alert stage.

DialogStatus Call # 6

```
  input:  none

  output: status:WORD - TRUE if Dialog Manager is active
```

DialogStatus returns TRUE if the Dialog Manager has been initialized. It would return FALSE before a DialogStartup and after a DialogShutDown.

Creating and Disposing of Dialogs

NewModalDialog Call # 10

```
  input:  dBoundsRect:LONG  pointer to the window bounds rectangle
          dVisible:WORD     TRUE if dialog is visible, FALSE if not
          dRefCon:LONG      any value you'd like to associate with
                            the dialog's window.

  output: theDialog:LONG    pointer to dialog port, zero if error.
```

NewModalDialog creates a modal dialog as specified by its parameters and returns a pointer to the port of the new dialog. It allocates memory for it.

dBoundsRect, a rectangle given in global coordinates, determines the dialog window's size and location. Remember that the top coordinate of this rectangle should be at least 25 points below the top of the screen for a modal dialog, to allow for the menu bar.

If the dVisible parameter is TRUE, the dialog window is drawn on the screen. If it's FALSE, the window is initially invisible and may later be shown with a call to the Window Manager procedure ShowWindow.

Note: NewModalDialog generates an update event for the entire window contents, so the items aren't drawn immediately. It allows you to do some processing on the items before to draw them. However, if you change the value of an item or its text, the Control Manager draws the item immediately. If you find that all the items should be drawn at the same time, try making the dialog invisible initially and then calling ShowWindow on the dialog to show it.

RefCon is the dialog window's reference value, which the application may store into and access for any purpose.

NewModalDialog sets the font of the dialog window's grafPort to the system font or, if you previously called SetDAFont, to the specified font. It also sets the dialog type in the dialog record to Modal_Type.

NewModelessDialog   Call # 11

```
  input:  dBoundsRect:LONG  pointer to window bounds rectangle.
          dTitle:LONG       pointer to string for dialog's title,
                            zero if no title.
          dBehind:LONG      pointer to window the dialog should be
                            behind.
          dFlag:WORD        Bit  vector  describing  the  dialog's
                            frame.
          dRefCon:LONG      any value you'd like to associate with
                            the dialog's window.
          dFullSize:LONG    pointer to RECT to be used as content's
                            zoomed size.

  output: theDialog:LONG    pointer to dialog port, zero if error.
```

Like NewModalDialog (above), NewModelessDialog creates a dialog as specified by its parameters and returns a pointer to the new dialog. Instead of making a modal dialog, NewModelessDialog creates a modeless dialog, as described under "Dialog and Alert Windows".

The dBoundsRect, dVisible and dRefCon parameters have the same value as in NewModalDialog.

dTitle is the title of the modeless dialog box.

The dBehind parameter specifies the window behind which the dialog window is to be placed on the desktop. Pass –1 ($FFFFFFFF) to bring up the dialog window in front of all other windows.

dFlag allows you to describe the window's frame of the dialog (close box, ...) .

dFullSize is a pointer to a rectangle describing the size and location of the dialog after the dialog is zoomed in.

```
GetNewModalDialog                 Call # 50

  input:  DialogTemplate:LONG pointer to a dialog template

  output: theDialog:LONG      pointer to dialog port, zero if error
```

GetNewModalDialog (like NewModalDialog) creates a modal dialog and returns a pointer to the port of the new dialog. But, instead of getting its parameters from the stack, it gets them from a template whose definition follows:

```
  DialogTemplate:

      BoundsRect:RECT        dialog bounds rectangle
      Visible:WORD           TRUE if dialog is to be visible
      RefCon:LONG            any value you want (application-use)
      Item1:LONG             pointer to first item's template
      Item2:LONG             pointer to second item's template
      ...
      ItemN:LONG             pointer to last item's template
      Terminator:LONG ZERO   item list terminated by a nil pointer.
```

The beginning of a dialog template contains the same values you would pass to NewModalDialog, except that BoundsRect is the actual rectangle, not a pointer.

The item1, item2,... itemN fields are pointers to item templates for each of the items you want to figure in the dialog. The last pointer must be 0 to signal the end of the list.

```
CloseDialog                       Call # 12

  input:  theDialog:LONG    pointer to dialog port

  output: none
```

CloseDialog removes theDialog's window from the screen and deletes it from the window list, just as when the Window Manager procedure CloseWindow is called. It releases the memory occupied by the following:

- The data structures associated with the dialog window (such as the window's structure, content, and update regions).
- All the items in the dialog (except for pictures and icons, which might be shared resources), and any data structures associated with them. For example, it would dispose of the region occupied by the thumb of a scroll bar, or a similar region for some other control in the dialog.

Creating and removing items:

NewDItem Call # 13

input:	theDialog:LONG	pointer to dialog this item belongs to.
	ItemID:WORD	item identifier for all item-related dialog manager calls.
	ItemRect:LONG	pointer to the display rectangle.
	ItemType:WORD	Button, Check, UserCtl, StatText, EditLine,PicItem,UserItem...
	ItemDescr:LONG	stringptr, textptr, procptr, iconhandle or pichandle.
	ItemValue:WORD	init value, text length, max length, 0, or any other value.
optional	ItemFlag:WORD	includes visible/invisible flag (0 for default flag).
optional	ItemColor:LONG	pointer to item's default color table (0 for default).

output: none.

Adds a new item to the dialog's item list.

The possible item types are: ButtonItem, CheckItem, RadioItem, ScrollBarItem, UserCtlItem, StatText, LongStatText, EditLine, IconItem, PicItem, UserItem.

If you add ItemDisable to the ItemType, the dialog manager will handle the actions on this disabled item without reporting anything to te application.

For a **Button** item [, **Check** item, **Radio** item], the itemDescr parameter is a pointer to the title of the button [, check box, radio button] and the itemValue is the initial value of the control (useful for check boxes and radio buttons).

For a **StatText** item, the itemDescr parameter is a pointer to a string containing the static text and the itemValue is not used. You can have several lines of text in the same item by inserting carriage returns (ASCII 13=$0D) inside the string. Here is an example of a typical string you would use for a StatText item:

```
StaticStr   dc   i1'EndStaticStr-StaticStr-1'
            dc   c'Do you want to save',h'0D'
            dc   c'before quitting?'
EndStaticStr anop
```

For a one line static text item, you can use the macro STR:

```
StaticStr   str 'File not found'
```

For a **LongStatText** item, the itemDescr parameter is a pointer to the beginning of the text ending and the itemValue is the word length of the text (0 to 32767). Here is an example of typical itemDescr and itemValue parameters you would use for a LongStatText item:

itemDescr is a pointer to the following text:

```
myLongText dc   c'This is a really very...',h'0D'
           dc   c'very... very...',h'0D'
           ...
           dc   c'long text, that contains',h'0D'
           dc   c'more than 255 characters',h'0D'
           dc   c'so that I need a LongStatText',h'0D'
           dc   c'item to print it in a single item'
EndLongText anop
```

and itemValue is: `EndLongText-myLongText`

For an **EditLine** item, the itemDescr parameter is a pointer to the default string containing the default text that first appears in the item when the dialog comes up and itemValue is the maximum allowed length of the editable string (0 to 255). Here is an example of typical itemDescr and itemValue parameters you would use for an EditLine item:

itemDescr is a pointer to the following string:

```
EditLStr    dc   i1'EndEditLStr-EditLStr-1' ; default string
            dc   c'Untitled'
EndEditLStr anop
```

and itemValue is: `15 (maximum length for a ProDos file name)`

If you pass zero for itemDescr, the line will have no default text in it.

If the item is the first EditLine item to be created, it will be the current active EditLine item and the default text (if there is any) will be selected.

For a **ScrollBar** item, the itemDescr is a pointer to a special action procedure that will be called during initialization time and scrolling. This procedure will be able, for example, to change the appearance of different items in the dialog in real-time, while the user is scrolling the scroll bar and without reporting anything to the application. In fact, if the scrollbar item is disabled, the application will not even know that the user clicked in it!

The definition of a Dialog ScrollBar Action Procedure follows:

MyDialogScrollBar

```
  input:    command:WORD       see list of possible commands below.
            dialog:LONG        dialog the scroll bar is in.
            ScrollBarID:WORD   item ID of scroll bar.

  output:   result:WORD        depends on command (see below).
```

Command	**Result**	**Comments**
GetInitView (1)	init view	view size at creation (called before control is allocated)
GetInitTotal (2)	init total	total size at creation (called before control is allocated)
GetInitValue (3)	starting value	value at creation (called before control is allocated)
ScrollLineUp (4)	new value	scroll one line up and return new scroll bar value
ScrollLineDown (5)	new value	scroll one line down and return new scroll bar value
ScrollPageUp (6)	new value	scroll one page up and return new scroll bar value
ScrollPageDown (7)	new value	scroll one page down and return new scroll bar value
ScrollThumb (8)	new value	get thumb position, scroll to that position and return new correct value (usually the same).

For the first three calls, do not make any reference to the scroll bar control because these calls are made before to allocate the control.

The calls from ScrollLineUp to ScrollPageDown should first call GetItemValue on ScrollBarID to get the previous value of the scroll bar, then do some changes (like changing an icon or the text of a StatText item, or adding or removing items from the dialog), and finally returns the new value of the scrollbar.

For ScrollThumb, you should first call GetItemValue on ScrollBarID. GetItemValue returns the new thumb position. You can then do whatever changes you want to do, and then returns either the value you got from GetItemValue or any value you find suitable.

Your Dialog ScrollBar Action procedure will be called by NewDItem just before to create a ScrollBar item and by ModalDialog when the user clicks in a ScrollBar item.

Note that ModalDialog will set the new scrollbar value according to the result returned by your procedure.

For an **Icon** item, itemDescr is a handle to an icon and itemValue is not used. The icon record contains the following fields:

```
iconRect    equ 0          ; bounds rect (width is multiple of 8)
iconImage   equ iconRect+8 ; pixel image (icon bitmap)
```

For a **Picture** item, itemDescr is a picture handle *** almost defined *** and itemValue is not used.

For a **UserControl** item, itemDescr is a pointer to a control definition procedure, as defined in the Control Manager ERS, and itemValue is the initial value of the control.

For a **User** item, itemDescr is a pointer to an item definition procedure and itemValue is not used. The definition of an item definition procedure follows:

```
MyItem

  input:   theDialog:LONG    pointer to the dialog's grafport
           itemID:WORD       ID of item to draw

  output:  none
```

The procedure for a UserItem draws the item; for example, if the item is a clock, it will draw the clock with the current time displayed. When this procedure is called, the current port will have been set by the Dialog Manager to the dialog window's grafPort.

TheDialog is a pointer to the dialog window; in case the procedure draws in more than one dialog, this parameters tells it which one to draw in.

ItemID is the item ID; in case the procedure draws more than one item, this parameter tells it which one to draw.

GetNewDItem Call # 51

```
  input:  theDialog:LONG       pointer to dialog port
          ItemTemplate:LONG    pointer to an item template

  output: none
```

GetNewDItem (like NewDItem) adds a new item to the dialog's item list. But, instead of getting its parameters from the stack, it gets them from a template whose definition follows:

```
  ItemTemplate:

      itemID:WORD           Number uniquely identifying the item
      itemRect:RECT         display rectangle, in local coordinates
      itemType:WORD         Type of item (Button, Check, Scroll...)
      itemDescr:LONG        Item Descriptor
      itemValue:WORD        Item Value
      itemFlag:WORD         Bit vector flag (0 for default)
      itemColor:LONG        Pointer to color table (0 for default)
```

Most of the item template fields are the same as those you would pass to NewDItem, except:

- ItemRect contains the actual display rectangle, not a pointer to it.
- The dialog that will contain the item is not specified in the template. This allows you to have dialog-independent items and repeat them among several dialogs (useful for OK, Cancel,... buttons).

RemoveItem Call # 14

```
  input:  theDialog:LONG     pointer to dialog port
          ItemID:WORD        ID of item to be removed

  output: none.
```

Removes the given item from the dialog and erases it from the screen.

Handling Dialog Events

```
ModalDialog                              Call # 15

  input:   filterProc:LONG    pointer to a filter procedure to be
                              called repeatdly

  output:  itemHit:WORD       ID of item hit.
```

Call ModalDialog after creating a modal dialog and bringing up its window in the frontmost plane. If the front window is a modaldialog, ModalDialog repeatedly gets and handles events in the dialog's window; after handling an event involving an enabled dialog item, it returns with the item ID in itemHit. Normally you'll then do whatever is appropriate as a response to an event in that item.

Note: If the front window is not a modal dialog (for instance, if it is a regular window or a modeless dialog), modaldialog returns immediately with itemHit set to 0.

ModalDialog gets each event by calling the Event Manager function GetNextEvent. If the event is a mouse-down event outside the content region of the dialog window, ModalDialog emits sound number 1 (which should be a single beep) and gets the next event; otherwise, it filters and handles the event as described below.

Note: Once before getting each event, ModalDialog calls SystemTask *** not called yet ***, a Desk Manager procedure that must be called regularly so that desk accessories will work properly.

The filterProc parameter determines how events are filtered. If it's NIL, the standard filterProc function is executed; this causes ModalDialog to return the ID of the default button (1 usually) in itemHit if the Return key is pressed and supports the Apple-X/C/V commands for Cut/Copy/Paste operations inside the dialog. If filterProc isn't NIL, ModalDialog filters events by executing the function it points to. Your filterProc function should have three parameters and return a Boolean value. For example, this is how it would be declared if it were named MyFilter:

```
MyFilter

  input:   theDialog:LONG     pointer to the dialog port.
           theEvent:LONG      pointer to the Event.
           itemHit:LONG       pointer to itemHit.

  output:  result:WORD        TRUE if must return.
```

A function result of FALSE tells ModalDialog to go ahead and handle the event, which either can be sent through unchanged or can be changed to simulate a different event. A function result of TRUE tells ModalDialog to return immediately rather than handle the event; in this case, the filterProc function sets itemHit to the item number that ModalDialog should return.

Note: If you set the bit 31 of the filterProc parameter to 1 before passing it to ModalDialog, the standard filter procedure will also be called after your filter procedure. It allows you to define a custom filter procedure and still get the benefits of the Cut/Copy/Paste feature and the Return alternative for the default button, for consistency with the Apple User Interface Guidelines.

You can use the filterProc function, for example, to treat a typed character in a special way (such as ignore it, or make it have the same effect as another character or as clicking a button); in this case, the function would test for a key event with that character. As another example, suppose the dialog box contains a userItem whose procedure draws a clock with the current time displayed. The filterProc function can call that procedure and return FALSE without altering the current event.

If you want the filter procedure to handle a special event and prevent ModalDialog from handling it, but without actually leaving ModalDialog, change the **what** field of the Event Record to **nullEvent** and returns FALSE.

ModalDialog handles the events for which the filterProc function returns FALSE as follows:

- In response to an activate or update event for the dialog window, ModalDialog activates or updates the window.
- If the mouse button is pressed in an EditLine item, ModalDialog responds to the mouse activity as appropriate (displaying an insertion point or selecting text). If a key event occurs without the Apple key held down and there's an EditLine item, text entry and editing are handled in the standard way for such items. If the Apple key is being held down, the typed character does not go to LineEdit except for left and right arrows. In either case, ModalDialog returns if the EditLine item is enabled or does nothing if it's disabled. If a key-down event occurs when there's no EditLine item, ModalDialog does nothing.
- If the mouse button is pressed in a standard or user control, ModalDialog calls the Control Manager function TrackControl. If the mouse button is released inside the control and the control is enabled, ModalDialog returns; otherwise, it does nothing.
- If the mouse button is pressed in a scroll bar item, ModalDialog calls the Control Manager function TrackControl with a special action procedure that calls your Dialog ScrollBar Action procedure.
- If the mouse button is pressed in any other enabled item in the dialog box, ModalDialog returns. If the mouse button is pressed in any other disabled item or in no item, or if any other event occurs, ModalDialog does nothing.

```
IsDialogEvent                        Call # 16

  input:  theEvent:LONG       pointer to the Event Record

  output: result:WORD         TRUE if theEvent is a Dialog Event.
```

If your application includes any modeless dialogs, call IsDialogEvent after calling the Event Manager function GetNextEvent. or the Window Manager function TaskMaster.

Warning: If your modeless dialog contains any EditLine items, you must call IsDialogEvent (and then DialogSelect) even if GetNextEvent returns FALSE; otherwise your dialog won't receive null events and the caret won't blink.

Pass the current event in theEvent. IsDialogEvent determines whether theEvent needs to be handled as part of a dialog. If theEvent is an activate or update event for a dialog window, a mouse-down event in the content region of an active dialog window, or any other type of event when a dialog window is active, IsDialogEvent returns TRUE; otherwise, it returns FALSE.

When FALSE is returned, just handle the event yourself like any other event that's not dialog-related. When TRUE is returned, you'll generally end up passing the event to DialogSelect for it to handle (as described below), but first you should do some additional checking:

- In special cases, you may want to bypass DialogSelect or do some preprocessing before calling it. If so, check for those events and respond accordingly.

For cases other than these, pass the event to DialogSelect for it to handle.

```
DialogSelect                 Call # 17

  input:  theEvent:LONG      pointer to the Event Record
          theDialog:LONG     address of variable to store the dialog
                             pointer in it
          itemHit:LONG       pointer to itemHit

  output: result:WORD        TRUE if event involved an enabled item
```

You'll normally call DialogSelect when IsDialogEvent returns TRUE, passing in theEvent an event that needs to be handled as part of a modeless dialog. DialogSelect handles the event as described below. If the event involves an enabled dialog item, DialogSelect returns a function result of TRUE with the dialog pointer in theDialog and the item number in itemHit; otherwise, it returns FALSE with theDialog and itemHit undefined. Normally when DialogSelect returns TRUE, you'll do whatever is appropriate as a response to the event, and when it returns FALSE you'll do nothing.

If the event is an activate or update event for a dialog window, DialogSelect activates or updates the window and returns FALSE.

If the event is a key-down or auto-key event and the Apple key is held down, DialogSelect returns FALSE.

If the event is a mouse-down event in an EditLine item, DialogSelect responds as appropriate (displaying a caret at the insertion point or selecting text). If it's a key-down or auto-key event without the Apple key being held down and there's an EditLine item, text entry and editing are handled in the standard way. In either case, DialogSelect returns TRUE if the EditLine item is enabled or FALSE if it's disabled. If a key-down or auto-key event is passed when there's no EditLine item, DialogSelect returns FALSE.

Note: To treat a typed character in a special way (such as ignore it, or make it have the same effect as another character or as clicking a button), you need to check for a key-down event with that character before calling DialogSelect.

If the event is a mouse-down event in a control, DialogSelect calls the Control Manager function TrackControl. If the mouse button is released inside the control and the control is enabled, DialogSelect returns TRUE; otherwise, it returns FALSE.

If the event is a mouse-down event in any other enabled item, DialogSelect returns TRUE. If it's a mouse-down event in any other disabled item or in no item, or if it's any other event, DialogSelect returns FALSE.

Note: If the event isn't one that DialogSelect specifically checks for (if it's a null event, for example), and there's an EditLine item in the dialog, DialogSelect calls the LineEdit procedure LEIdle to make the caret blink.

```
DlgCut                                  Call # 18

  input:  theDialog:LONG    pointer to the dialog

  output: none
```

DlgCut checks whether theDialog has any EditLine items and, if so, applies the LineEdit procedure LECut to the currently selected EditLine item. You can call DlgCut to handle the editing command Cut when a modeless dialog window is active.

```
DlgCopy                                 Call # 19

  input:  theDialog:LONG    pointer to the dialog

  output: none
```

DlgCopy is the same as DlgCut except that it calls LECopy, for handling the Copy command.

```
DlgPaste                                Call # 20

  input:  theDialog:LONG    pointer to the dialog

  output: none
```

DlgPaste is the same as DlgCut except that it calls LEPaste, for handling the Paste command.

```
DlgDelete                               Call # 21

  input:  theDialog:LONG    pointer to the dialog

  output: none
```

DlgDelete is the same as DlgCut except that it calls LEDelete, for handling the Clear command.

```
DrawDialog                              Call # 22

  input:  theDialog:LONG    pointer to the dialog

  output: none
```

DrawDialog draws the contents of the given dialog box. Since DialogSelect and ModalDialog handle dialog window updating, this procedure is useful only in unusual situations. You would call it, for example, to display a dialog box that doesn't require any response but merely tells the user what's going on during a time-consuming process.

Invoking Alerts

```
Alert                               Call # 23

  input:  AlertTemplate:LONG  pointer to an alert template
          filterProc:LONG     pointer to filter used by ModalDialog

  output: itemHit:WORD        ID of item Hit.
```

This function invokes the alert defined by the alert template. It calls the current sound procedure, if any, passing it the sound number specified in the alert template for this stage of the alert. If no alert box is to be drawn at this stage, Alert returns a function result of –1; otherwise, it creates and displays the alert window for this alert and draws the alert box.

Alert gets its parameters for an alert template. The definition of an alert template is as follows:

```
  AlertTemplate:

      BoundsRect:RECT        alert bounds rectangle
      AlertID:WORD           number uniquely identifying the alert
      stage1:BYTE            first stage of alert
      stage2:BYTE            second stage of alert
      stage3:BYTE            third stage of alert
      stage4:BYTE            fourth stage of alert
      Item1:LONG             pointer to first item's template
      Item2:LONG             pointer to second item's template
      ...
      ItemN:LONG             pointer to last item's template
      Terminator:LONG ZERO   item list terminated by a nil pointer.
```

A stage byte is a bit vector containing the following bit fields:

```
      Bits 0-2  : Sound Number to emit at this stage (0 to 3)
      Bits 3-5  : Unused
      Bit  6    : Default button ID minus 1 (only 1 or 2).
      Bit  7    : Flag indicating if the alert should be drawn.
```

Note: Alert creates the alert window by calling NewModalDialog and GetNewDItem for each item in the alert, and does the rest of its processing by calling ModalDialog.

Alert repeatedly gets and handles events in the alert window until an enabled item is clicked, at which time it returns the item number. Normally you'll then do whatever is appropriate in response to a click of that item.

Alert gets each event by calling the Event Manager function GetNextEvent. If the event is a mouse-down event outside the content region of the alert window, Alert emits sound number 1 (which should be a single beep) and gets the next event; otherwise, it filters and handles the event as described below.

The filterProc parameter has the same meaning as in ModalDialog (see above). If it's NIL, the standard filterProc function is executed, which makes the Return key have the same effect as clicking the default button. If you specify your own filterProc function and want to retain this feature, you must set the bit 31 of the filterProc parameter to 1. You can find out what the current default button is by calling GetDefButton on the dialog pointer for the alert passed to your filter procedure.

Alert handles the events for which the filterProc function returns FALSE as follows:

- If the mouse button is pressed in a control, Alert calls the Control Manager procedure TrackControl. If the mouse button is released inside the control and the control is enabled, Alert returns; otherwise, it does nothing.

- If the mouse button is pressed in any other enabled item, Alert simply returns. If it's pressed in any other disabled item or in no item, or if any other event occurs, Alert does nothing.

Before returning to the application with the item number, Alert removes the alert box from the screen. (It disposes of the alert window and its associated data structures, the item list, and the items.)

Note: The Alert function's removal of the alert box would not be the desired result if the user clicked a check box or radio button; however, normally alerts contain only static text, icons, pictures, and buttons that are supposed to make the alert box go away. If your alert contains other items besides these, consider whether it might be more appropriate as a dialog.

```
StopAlert                        Call # 24         *** without icon ***

  input:  AlertTemplate:LONG     pointer to an alert template
          filterProc:LONG        pointer to filter used by ModalDialog

 output:  itemHit:WORD           ID of item Hit.
```

StopAlert is the same as the Alert function (above) except that before drawing the items of the alert in the alert box, it draws the Stop icon in the top left corner of the box (within the rectangle (10,20)(42,52)). The Stop icon has the following ID:

```
stopIcon  equ   0
```

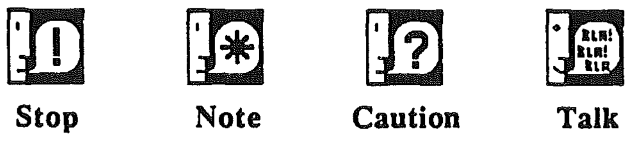

Figure 7. Standard Alert Icons

NoteAlert Call # 25 **** without icon ****

```
  input:  AlertTemplate:LONG  pointer to an alert template
          filterProc:LONG     pointer to filter used by ModalDialog

  output: itemHit:WORD        ID of item Hit.
```

NoteAlert is like StopAlert except that it draws the Note icon, which has the following ID:

```
noteIcon  equ  1
```

CautionAlert Call # 26 **** without icon ****

```
  input:  AlertTemplate:LONG  pointer to an alert template
          filterProc:LONG     pointer to filter used by ModalDialog

  output: itemHit:WORD        ID of item Hit.
```

CautionAlert is like StopAlert except that it draws the Caution icon, which has the following ID:

```
cautionIcon equ  2
```

TalkAlert Call # 28 **** without speech (yet) ****

```
  input:  AlertTemplate:LONG  pointer to an alert template
          filterProc:LONG     pointer to filter used by ModalDialog

  output: itemHit:WORD        ID of item Hit.
```

TalkAlert is like StopAlert except that it draws the Talk icon, which has the following ID:

```
talkIcon    equ  3
```

and it then calls the Sound Tools to actually SPEAK the alert with a synthetic voice.

Manipulating Items in Dialogs and Alerts

ParamText Call # 27

```
input:  param0:LONG        pointer to string ^0 (zero=no change)
        param1:LONG        pointer to string ^1 (zero=no change)
        param2:LONG        pointer to string ^2 (zero=no change)
        param3:LONG        pointer to string ^3 (zero=no change)

output: none
```

ParamText provides a means of substituting text in statText items: param0 through param3 will replace the special strings '^0' through '^3' in all statText items in all subsequent dialog or alert boxes. Pass empty strings for parameters not used.

You may pass NIL for parameters not used or for strings that are not to be changed.

For example, if the text is defined as 'Cannot open document ^0' and docName is a string variable containing a document name that the user typed, you can call ParamText(docName,' ',' ',' ').

GetControlItem Call # 30

```
input:  theDialog:LONG       pointer to the dialog port
        itemID:WORD          Unique number identifying the item.

output: theControl:LONG      handle to the control.
```

Given the ID of an item, GetControlItem returns a handle to the control for this item. You can then make calls the Control Manager to change the behavior of this item.

Warning: Be very careful with GetControlItem, because, by using directly the Control Manager, you bypass the Dialog Manager and could destroy some datas used by the Dialog Manager. It is however safe to use GetControlItem on standard controls (like Buttons, Check Boxes and Radio Buttons). It is a little bit less safe to use it with dialog scroll bars. And it is definitely unsafe to use it with text items. Whatever you do, do not change the CtrlRefCon field in the Control Record of any control.

Note: A list of Dialog Manager calls are provided to change the attributes of items. Whenever possible, it is highly recommended to use these calls instead of Control Manager calls.

GetIText Call # 31

```
input:  theDialog:LONG     pointer to the dialog
        itemID:WORD        ID of item in dialog
        theString:LONG     pointer to a string to put the text in.

output: none
```

Given the ID of a StatText or EditLine item in a dialog box, GetIText returns the text of the item in the text parameter.

Note: The space for the string must be allocated (must exists) before to call GetIText.

SetIText Call # 32

```
input:  theDialog:LONG     pointer to the dialog
        itemID:WORD        ID of item in dialog
        theString:LONG     pointer to the new text string.

output: none
```

Given the ID of a StatText or EditLine item in a dialog box,, SetIText sets the text of the item to the specified text and draws the item. For example, suppose the exact content of a dialog's text item cannot be determined until the dialog is created, but the display rectangle is already defined, call SetIText with the desired text.

SelIText Call # 33

```
input:  theDialog:LONG     pointer to the dialog
        itemID:WORD        ID of item in dialog
        startSel:WORD      start of selection
        endSel:WORD        end of selection

output: none
```

Given a pointer to a dialog and the item ID of an EditLine item in the dialog box, SelIText does the following:

- If the item contains text, SelIText sets the selection range to extend from character position startSel up to but not including character position endSel. The selection range is inverted unless startSel equals endSel, in which case a blinking vertical bar is displayed to indicate an insertion point at that position.

- If the item doesn't contain text, SelIText simply displays the insertion point.

For example, if the user makes an unacceptable entry in the EditLine item, the application can put up an alert box reporting the problem and then select the entire text of the item so it can be replaced by a new entry. (Without this procedure, the user would have to select the item before making the new entry.)

Note: You can select the entire text by specifying 0 for startSel and 32767 for endSel. For details about selection range and character position, see the LineEdit ERS Manual.

```
GetItemType                          Call # 38

  input:  theDialog:LONG    pointer to the dialog
          itemID:WORD       ID of item in dialog

  output: itemType:WORD     type of item, including ItemDisable bit
```

GetItemType returns the type of the specified item (ButtonItem, RadioItem, StatText,...). If the item is disabled (from the dialog manager point of view), the returned value is the type plus ItemDisable.

```
SetItemType                          Call # 39

  input:  itemType:WORD     type of item, including ItemDisable bit
          theDialog:LONG    pointer to the dialog
          itemID:WORD       ID of item in dialog

  output: none
```

SetItemType changes the specified item to the new desired type. If you want the item to be disabled add ItemDisable to itemType.

Note: SetItemType does not redraw the item. This allows you to change the type of several items, and then redraws all the changes at the same time.

Warning: Changing the type of an item can be very dangerous, since the structure of two items of different types can be very different. And what is the interest of changing the type of an item, except to change the ItemDisable status, which can be done easily by SetItemDisable.

```
GetItemBox                           Call # 40

  input:  theDialog:LONG    pointer to the dialog
          itemID:WORD       ID of item in dialog
          itemBox:LONG      pointer to space to store the rect in

  output: none
```

GetItemBox returns the display rectangle of the specified item in the variable itemBox.

```
SetItemBox                         Call # 41

  input:  theDialog:LONG     pointer to the dialog
          itemID:WORD        ID of item in dialog
          itemBox:LONG       pointer to the new display rectangle

  output: none
```

SetItemBox changes the display rectangle of the specified item to itemBox.

```
GetFirstItem                       Call # 42

  input:  theDialog:LONG     pointer to the dialog

  output: firstItem:WORD     ID of first item in dialog, 0 if none
```

GetFirstItem returns the ID of the first item in the dialog. If there is no item in the dialog (just after NewModalDialog or NewModelessDialog, for example), GetFirstItem returns zero.

Warning: Since there may be some collisions between an item of ID zero and no item at all, you should not have any item with an ID of zero.

```
GetNextItem                        Call # 43

  input:  theDialog:LONG     pointer to the dialog
          itemID:WORD        ID of item in dialog

  output: nextItem:WORD      ID of next item in dialog, 0 if no more
```

GetNextItem returns the ID of the next item in the dialog after itemID. If itemID is the last item in the dialog, GetNextItem returns zero.

Warning: Since there may be some collisions between an item of ID zero and no item at all, you should not have any item with an ID of zero.

```
GetDefButton                       Call # 55

  input:  theDialog:LONG     pointer to the dialog

  output: DefButID:WORD      ID of dialog default button, or zero
```

GetDefButton returns the ID of the default button item in the dialog. If the dialog does not contain any default button, GetDefButton returns zero.

SetDefButton Call # 56

```
  input:  DefButID:WORD      ID of new default button
          theDialog:LONG     pointer to the dialog

  output: none
```

SetDefButton sets the ID of the default button to DefButID.

Warning: DefButID must be the ID of a button item. Also, if you do not call SetDefButton to specify explicitely which button is the default button, the Dialog Manager assumes that the item of ID 1 is the default button. So, be sure that either there is no item with an ID 1 or it is a button.

GetItemFlag Call # 44

```
  input:  theDialog:LONG     pointer to the dialog
          itemID:WORD        ID of item in dialog

  output: itemFlag:WORD      Bit vector Flag of item
```

GetItemFlag returns the bit vector flag for the specified item. For standard controls, itemFlag is equivalent to the CtrlFlag field in the Control Structure. For the other types of items, itemFlag may have special meanings. For example, for a picture item, a bit in the flag specifies if the picture should be clipped or stretched to fit in the display rectangle. For an icon item, the flag tells if the icon is a "Finder-type" icon or an "Alert-Type" icon (icon with a mask versus icon without a mask).

SetItemFlag Call # 45

```
  input:  itemFlag:WORD      new flag for item
          theDialog:LONG     pointer to the dialog
          itemID:WORD        ID of item in dialog

  output: none
```

SetItemFlag changes the bit vector flag of the specified item to the new desired flag. See GetItemFlag for more details.

Note: SetItemFlag does not redraw the item. So, do not use SetItemFlag to change the invisibility status. Use the HideDItem and ShowDItem procedures instead.

```
GetItemValue                  Call # 46

  input:  theDialog:LONG    pointer to the dialog
          itemID:WORD       ID of item in dialog

  output: itemValue:WORD    Current value of item
```

GetItemValue returns the current value of the specified item. For standard controls, itemValue is the current value of the control. For the other types of items, itemValue may have special meaning ***more details to come***.

```
SetItemValue                  Call # 47

  input:  itemValue:WORD    new value
          theDialog:LONG    pointer to the dialog
          itemID:WORD       ID of item in dialog

  output: none
```

SetItemValue sets the value of the specified item to the new desired value and redraws the item. See GetItemValue for more details.

```
GetItemColor                  Call # 48

  input:  theDialog:LONG    pointer to the dialog
          itemID:WORD       ID of item in dialog

  output: itemColor:LONG    Pointer to current color table
```

GetItemColor returns a pointer to the current color table for the specified item ***more details to come***.

```
SetItemColor                  Call # 49

  input:  itemColor:LONG    pointer to new color table
          theDialog:LONG    pointer to the dialog
          itemID:WORD       ID of item in dialog

  output: none
```

SetItemColor sets the color table of the specified item to the new desired color table. See GetItemColor for more details.

```
GetAlertStage                     Call # 52

  input:  none

  output: alertStage:WORD   current stage of the alert
```

GetAlertStage returns the stage of the last occurrence of an alert, as a number from 0 to 3.

```
ResetAlertStage                   Call # 53

  input:  none

  output: none
```

ResetAlertStage resets the stage of the last occurrence of an alert so that the next occurrence of that same alert will be treated as its first stage. This is useful, for example, when you've used ParamText to change the text of an alert such that from the user's point of view it's a different alert.

```
DefaultFilter                     Call # 54

  input:  theDialog:LONG    pointer to the dialog port
          theEvent:LONG     pointer to the event
          itemHit:LONG      pointer to itemHit

  output: result:WORD       TRUE if must return
```

DefaultFilter calls the standard default filter used by ModalDialog or Alert when no user filter procedure is specified. Given a pointer to an event involving dialog items, DefaultFilter filters the Apple-X, Apple-C, Apple-V keys to make them cut, copy and paste, and interprets the Return key as a click in the default button.

DefaultFilter returns TRUE in result if the default button has been clicked in and itemHit contains its ID number, and if a Cut/Copy/Paste operation has been made on an enabled EditLine item.

```
HideDItem                         Call # 34

  input:  theDialog:LONG    pointer to the dialog port
          itemID:WORD       ID of item to hide

  output: none
```

HideDItem erases from the dialog the specified item. The item is not removed from the item list. It can be shown again by just calling ShowDItem.

If the item is already invisible, HideDItem does nothing.

```
ShowDItem                       Call # 35

  input:  theDialog:LONG     pointer to the dialog port
          itemID:WORD        ID of item to show

  output: none
```

ShowDItem makes visible an item that has been created invisible or has been hidden by HideDItem.

If the item is already visible, ShowDItem does nothing.

```
FindDItem                       Call # 36

  input:  theDialog:LONG     pointer to the dialog port
          thePoint:LONG      point in global coordinates (passed as
                             a long word)

  output: itemHit:WORD       ID of item located at thePoint, or zero
```

FindDItem returns the ID of the item located at the specified point in the specified dialog. If there is no item at this location or if thePoint is outside the dialog, FindDItem returns zero.

thePoint must be in global coordinates.

```
UpdateDialog                    Call # 37

  input:  theDialog:LONG     pointer to the dialog port
          UpdateRgn:LONG     handle to the region to update

  output: none
```

UpdateDialog redraws only the part of the dialog that is in the specified update region.

If UpdateRgn was part of a region to update in an up-coming update event for the dialog, you should call ValidRgn to prevent the Dialog Manager to redraw this particular region twice.

```
DisableItem                     Call # 57

  input:  theDialog:LONG     pointer to the dialog port
          itemID:WORD        ID of item to disable

  output: none
```

DisableItem disables the specified item (**Warning**: disabled is different from deactivated). If the item is already disabled, DisableItem does nothing.

```
EnableItem                      Call # 58

  input:  theDialog:LONG    pointer to the dialog port
          itemID:WORD       ID of item to enable

  output: none
```

EnableItem enables the specified item (**Warning**: enabled is different from active). If the item is already enabled, EnableItem does nothing.

SUMMARY OF THE DIALOG MANAGER

Constants

```
*
* Booleans
*

The size of a boolean is a WORD. Its possible values are:

FALSE: 0 (ZERO)
TRUE:  any non-zero value

*
* Errors returned by the Dialog Manager
*    (Note that the Dialog Manager may also return errors coming
*     from the Window Manager, Control Manager, Memory Manager
*     and LineEdit)
*
BadItemType     equ  $150A  ; item is not of appropriate type
NewItemFailed   equ  $150B  ; creation of item failed
ItemNotFound    equ  $150C  ; item does not exist

*
* Item Types
*
ButtonItem      equ  10     ; standard button control
CheckItem       equ  11     ; standard check box control
RadioItem       equ  12     ; standard radio button control
ScrollBarItem   equ  13     ; special scrollbar control for dialogs
UserCtlItem     equ  14     ; custom control
StatText        equ  15     ; static text
LongStatText    equ  16     ; long static text (up to 32767 chars)
EditLine        equ  17     ; editable line of text (dialog only)
IconItem        equ  18     ; icon
PicItem         equ  19     ; QuickDraw picture
UserItem        equ  20     ; custom item (dialog only)
ItemDisable     equ  $8000  ; add to any of above to disable

*
* Item IDs of standard OK and Cancel buttons
*
ok              equ  1
cancel          equ  2
```

```
*
* icon IDs of alert icons
*
stopIcon        equ   0
noteIcon        equ   1
cautionIcon     equ   2
talkIcon        equ   3

*
* structure of an icon
*
iconRect        equ   0             ; bounds rect (multiple of 8)
iconImage       equ   iconRect+8    ; pixel image (icon bitmap)

*
* Dialog ScrollBar Commands
*
GetInitView     equ   1        ; view size at creation
GetInitTotal    equ   2        ; total size at creation
GetInitValue    equ   3        ; value at creation
ScrollLineUp    equ   4        ; scroll one line up
ScrollLineDown  equ   5        ; scroll one line down
ScrollPageUp    equ   6        ; scroll one page up
ScrollPageDown  equ   7        ; scroll one page down
ScrollThumb     equ   8        ; scroll to thumb position
```

Data Types

```
TYPE
  DialogPtr = WindowPtr;
```

ItemTemplate:

itemID:WORD	Number uniquely identifying the item
itemRect:RECT	display rectangle, in local coordinates
itemType:WORD	Type of item (Button, Check, Scroll...)
itemDescr:LONG	Item Descriptor
itemValue:WORD	Item Value
itemFlag:WORD	Bit vector flag (0 for default)
itemColor:LONG	Pointer to color table (0 for default)

DialogTemplate:

```
    BoundsRect:RECT        dialog bounds rectangle
    Visible:WORD           TRUE if dialog is to be visible
    RefCon:LONG            any value you want (application-use)
    Item1:LONG             pointer to first item's template
    Item2:LONG             pointer to second item's template
    ...
    ItemN:LONG             pointer to last item's template
    Terminator:LONG ZERO   item list terminated by a nil pointer.
```

AlertTemplate:

```
    BoundsRect:RECT        alert bounds rectangle
    stage1:BYTE            first stage of alert
    stage2:BYTE            second stage of alert
    stage3:BYTE            third stage of alert
    stage4:BYTE            fourth stage of alert
    Item1:LONG             pointer to first item's template
    Item2:LONG             pointer to second item's template
    ...
    ItemN:LONG             pointer to last item's template
    Terminator:LONG ZERO   item list terminated by a nil pointer.
```

Stage Bit vector:

```
    Bits 0-2  : Sound Number to emit at this stage (0 to 3)
    Bits 3-5  : Unused
    Bit   6   : Default button ID minus 1 (only 1 or 2).
    Bit   7   : Flag indicating if the alert should be drawn.
```