when the user clicks its close box. The next time the user invokes the Global Changes command, the dialog box is already available, in the same location and with the same text selected as when it was last used.

If you adjust the menus when you display a dialog box, be sure to return them to an appropriate state when you close the dialog box, as described in "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73.

Dialog Manager Reference

This section describes the data structure, routines, and resources that are specific to the Dialog Manager.

The "Data Structure" section shows the Pascal data structure for the dialog record, which the Dialog Manager creates and maintains. The "Dialog Manager Routines" section describes Dialog Manager routines for invoking alerts, creating and disposing of dialog boxes, manipulating items in alert and dialog boxes, and handling events in dialog boxes.

The "Application-Defined Routines" section describes routines that your application must supply when you need to create application-defined items in dialog boxes, to filter events that the Dialog Manager doesn't handle, and to define its own alert sounds.

The "Resources" section describes the dialog resource, the alert resource, the item list resource, the dialog color table resource, the alert color table resource, and the item color table resource. The summary sections that conclude this chapter include listings of the constants that define values for the item types in alert and dialog boxes, the OK and Cancel buttons in alert boxes, and the icons in note alert boxes, caution alert boxes, and stop alert boxes, along with the constants used by the `Gestalt` function for the Dialog Manager.

Data Structure

This section describes the dialog record. Your application doesn't need to create or use this record; rather, your application simply uses the appropriate Dialog Manager routines, creates any necessary resources, and then allows the Dialog Manager to create and use records of this data type as necessary. The dialog record is described here for completeness only.

The Dialog Record

To create an alert or a dialog box, you use a Dialog Manager routine—such as `Alert` or `GetNewDialog`—that incorporates information from your item list resource and from your alert resource or dialog resource into a data structure, called a *dialog record*, in memory. The Dialog Manager creates a dialog record, which is a data structure of type `DialogRecord`, whenever your application creates an alert or a dialog box. Your application generally should not create a dialog record or directly access its fields.

```
TYPE  DialogPtr      = WindowPtr;
      DialogPeek     = ^DialogRecord

      DialogRecord   =
      RECORD
         window:     WindowRecord;  {dialog window}
         items:      Handle;        {item list resource}
         textH:      TEHandle;      {current editable text item}
         editField:  Integer;       {editable text item number }
                                    { minus 1}
         editOpen:   Integer;       {used internally; reserved}
         aDefItem:   Integer;       {default button item number}
      END;
```

Field descriptions

`window`	The window record for the alert box or dialog box.
`items`	A handle to the item list resource for the alert or the dialog box.
`textH`	A handle to the current editable text item.
`editField`	The current editable text item.
`editOpen`	Used internally; reserved.
`aDefItem`	The item number of the default button.

Dialog Manager Routines

This section describes the routines for initializing the Dialog Manager, invoking alerts, creating and disposing of dialog boxes, manipulating items in alert and dialog boxes, and handling events in alert and dialog boxes.

Some Dialog Manager routines can be accessed using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, `GetDialogItem` is also available as `GetDItem`. Table 6-1 provides a mapping between the previous name of a routine and its new equivalent name.

Table 6-1 Mapping between new and previous names of Dialog Manager routines

New name	Previous name
`DialogCopy`	`DlgCopy`
`DialogCut`	`DlgCut`
`DialogDelete`	`DlgDelete`
`DialogPaste`	`DlgPaste`
`DisposeDialog`	`DisposDialog`
`FindDialogItem`	`FindDItem`

Table 6-1 Mapping between new and previous names of Dialog Manager routines (continued)

New name	Previous name
`GetAlertStage`	`GetAlrtStage`
`GetDialogItem`	`GetDItem`
`GetDialogItemText`	`GetIText`
`HideDialogItem`	`HideDItem`
`NewColorDialog`	`NewCDialog`
`ResetAlertStage`	`ResetAlrtStage`
`SelectDialogItemText`	`SelIText`
`SetDialogFont`	`SetDAFont`
`SetDialogItem`	`SetDItem`
`SetDialogItemText`	`SetIText`
`ShowDialogItem`	`ShowDItem`
`UpdateDialog`	`UpdtDialog`

Initializing the Dialog Manager

Before using the Dialog Manager, you must initialize—in order—QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit. The first Dialog Manager routine to call is the `InitDialogs` procedure, which initializes the Dialog Manager.

At your application's request, the Dialog Manager uses the system alert sound for signaling the user during various alert stages. For alerts, if you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure (described on page 6-144) and call the `ErrorSound` procedure to make it the current sound procedure.

By default, the Dialog Manager displays static text and editable text items in the system font. To make it easier to localize your application for use with worldwide versions of system software, you should not change the font. However, if you determine that it is imperative for your application to display static text or editable text in a font other than the system font, you can use the `SetDialogFont` procedure.

InitDialogs

Use the `InitDialogs` procedure to initialize the Dialog Manager.

```
PROCEDURE InitDialogs (resumeProc:  ResumeProcPtr);
```

`resumeProc`

A pointer to a procedure used by the System Error Handler in case a fatal system error occurs on a system that predates MultiFinder. For System 7, your application should set this parameter to `NIL`.

DESCRIPTION

Before using the Dialog Manager, you must initialize QuickDraw, the Font Manager, the Window Manager, the Menu Manager, and TextEdit, in that order. Then, to initialize the Dialog Manager, call `InitDialogs` once before all other Dialog Manager routines. The `InitDialogs` procedure does the following initialization:

- It saves the pointer passed in the `resumeProc` parameter. For System 7, your application should set the `resumeProc` parameter to `NIL`.
- It installs the system alert sound. To change the system alert sound, use the `ErrorSound` procedure.
- It passes empty strings to the `ParamText` procedure.

ErrorSound

To use your own alert sound instead of the system alert sound for signaling the user, use the `ErrorSound` procedure.

```
PROCEDURE ErrorSound (soundProc: SoundProcPtr);
```

`soundProc` A pointer to a procedure that generates the desired alert sounds.

DESCRIPTION

The Dialog Manager uses the system alert sound for signaling the user during various alert stages. The system alert sound, which is a sound resource stored in the System file, is played whenever system software or your application uses the Sound Manager procedure `SysBeep`. By changing the setting in the Sound control panel, the user can determine which sound is played. If you want to use sounds other than the system alert sound at various alert stages, write your own sound procedure and call the `ErrorSound` procedure to make it the current sound procedure.

SPECIAL CONSIDERATIONS

If you pass `NIL` in the `soundProc` parameter, the Dialog Manager neither plays sounds nor causes the menu bar to blink, and thus the user receives no signal.

SEE ALSO

See the description of `MyAlertSound` on page 6-144 for a discussion of how to write the sound procedure pointed to by the `soundProc` parameter. For examples of how to incorporate sound alerts into alert stages, see Listing 6-2 on page 6-21 and Listing 6-3 on page 6-22.

SetDialogFont

Although you generally should not change the font used in static and editable text items, you can do so with the `SetDialogFont` procedure. The `SetDialogFont` procedure is also available as the `SetDAFont` procedure.

```
PROCEDURE SetDialogFont (fontNum: Integer);
```

`fontNum` A font ID number. Do not rely on font number constants. Instead, use the Font Manager function `GetFNum` to find the font number to pass in this parameter.

DESCRIPTION

For subsequently created dialog and alert boxes, `SetDialogFont` sets the font of the dialog or alert box's graphics port to the specified font. If you don't call this procedure, the system font is used. The `SetDialogFont` procedure does not affect titles of controls, which are always displayed in the system font.

SPECIAL CONSIDERATIONS

There are a number of caveats regarding the `SetDialogFont` procedure.

First, the Standard File Package does not always properly calculate the position of the standard file dialog box once this procedure has been called; for example, the standard file dialog box may be partially obscured by a menu bar. Second, be aware that this procedure affects all static text and editable text items in all of the alert and dialog boxes you subsequently display. Third, `SetDialogFont` does not change the font for control titles. Fourth, you can't use `SetDialogFont` to change the font size or font style. Finally, and most importantly, your application will be much easier to localize if you always use the system font in your alert and dialog boxes and never use `SetDialogFont`.

SEE ALSO

See the chapter "Font Manager" in *Inside Macintosh: Text* for information about the `GetFNum` function.

Creating Alerts

To create an alert—consisting of an alert sound, an alert box, or both—use one of these functions: `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert`. The first three functions display, respectively, the note, caution, and stop alert icons (see Figure 6-3, Figure 6-4, and Figure 6-5) in the upper-left corner of the alert box. The `Alert` function allows you to display your own icon or to have no icon at all in the upper-left corner of your alert box.

These functions take descriptive information about the alert from an alert resource that you provide. When you call one of these functions, you pass it the resource ID of the alert resource and a pointer to an event filter function. These functions create a dialog record, play an alert sound, and display an alert box according to the alert stages that you specify in the alert resource.

You should specify a pointer to an event filter function when you call the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. You can use the same event filter function in most or all of your alert and modal dialog boxes.

If you need to find out the current alert stage—for example, to ensure that your application deactivates the frontmost window only if an alert box is to be displayed at that stage—use the `GetAlertStage` function. To change the current alert stage, use the `ResetAlertStage` procedure.

Your application does not dispose of alert boxes; the Dialog Manager does that for you automatically.

Alert

To display an alert box (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), you can use the `Alert` function. This function does not display a default icon in the upper-left corner of the alert box; you can leave this area blank, or you can specify your own icon in the alert's item list resource, which in turn is specified in the alert resource.

```
FUNCTION Alert (alertID: Integer;
                filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure.

DESCRIPTION

The `Alert` function creates the alert defined in the specified alert resource. The function calls the current alert sound procedure and passes it the sound number specified in the alert resource for the current alert stage. If no alert box is to be drawn at this stage, `Alert` returns –1; otherwise, it uses the `NewDialog` function to create and display the alert box. The default system window colors are used unless your application provides an alert color table resource with the same resource ID as the alert resource.

The `Alert` function uses the `ModalDialog` procedure, which repeatedly gets and handles most events for you. The `ModalDialog` procedure, in turn, 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 box, `ModalDialog` emits an error sound and gets the next event.

The `Alert` function continues calling `ModalDialog` until the user selects an enabled control (typically a button), at which time the `Alert` function removes the alert box from the screen and returns the item number of the selected control. Your application then responds as appropriate when the user clicks this item.

For events inside the alert box, `ModalDialog` passes the event to an event filter function before handling the event. The event filter function provides a secondary event-handling loop for events that `ModalDialog` doesn't handle. You specify a pointer to your event filter function in the `filterProc` parameter of the `Alert` function.

If you set the `filterProc` parameter to `NIL`, the Dialog Manager uses the standard event filter function, which behaves as follows:

- If the user presses the Return or Enter key, the event filter function returns `TRUE` and returns the item number for the default button.

However, your application should provide a simple event filter function that not only replicates this behavior but also

- returns `TRUE` and the item number for the Cancel button if the user presses Esc or Command-period
- updates your windows in response to update events (this also allows background windows to receive update events) and returns `FALSE`
- returns `FALSE` for all events that your event filter function doesn't handle

You can also use the event filter function to test for and respond to keyboard equivalents.

Unless the event filter function handles the event in its own way and returns `TRUE`, `ModalDialog` handles the event inside the alert box as follows:

- If the user presses the mouse button while the cursor is in a control, the Control Manager function `TrackControl` tracks the cursor. If the user releases the mouse button while the cursor is in an enabled control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the control's item number. (Generally, buttons should be the only controls you use in alert boxes.)
- If the user presses the mouse button while the cursor is in any enabled item other than a control, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` remove the alert box and return the item number. (Generally, button controls should be the only enabled items in alert boxes.)
- If user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` do nothing.

The `Alert` function uses the QuickDraw routine `SetPort` to make the alert box the current graphics port. It's not necessary for your application to call `SetPort` again before displaying alert boxes, because you can't draw into any other windows between the time you create an alert box and the time the Dialog Manager displays it.

SPECIAL CONSIDERATIONS

If you need to display an alert box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you will not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to an alert box that your application presents.

SEE ALSO

The `ModalDialog` procedure is described on page 6-135. See "Writing an Event Filter Function for Alert and Modal Dialog Boxes" beginning on page 6-86 for a discussion of how to write an event filter function. See "Creating Alert Sounds and Alert Boxes" beginning on page 6-18 for a discussion of alerts and alert stages. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about button titles and messages in alert boxes. Alert resources are described on page 6-150. Alert color table resources are described on page 6-157. The Dialog Manager uses the system alert sound as the error sound unless you change it by calling the `ErrorSound` procedure, described on page 6-104. See "Responding to Events in Alert Boxes" beginning on page 6-81 for a discussion of how to respond to events returned by the `Alert` function. See the chapter "Notification Manager" in *Inside Macintosh: Processes* for information about the Notification Manager.

The `NoteAlert`, `CautionAlert`, and `StopAlert` functions are identical to the `Alert` function, except that `NoteAlert` (described on page 6-110), `CautionAlert` (described on page 6-111), and `StopAlert` (described next) display icons in the upper-left corners of alert boxes.

StopAlert

To display an alert box with a stop icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `StopAlert` function.

```
FUNCTION StopAlert (alertID: Integer;
                    filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `StopAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `StopAlert` draws the stop icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The stop icon has the following resource ID:

```
CONST stopIcon = 0;  {stop icon}
```

By default, the Dialog Manager uses the standard stop icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a stop alert to inform the user that a problem or situation is so serious that the action cannot be completed. Stop alerts typically have only a single button (OK), because all the user can do is acknowledge that the action cannot be completed.

SEE ALSO

Figure 6-5 on page 6-9 illustrates the stop icon in a typical stop alert. Except that it includes a stop icon in the alert box, `StopAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

NoteAlert

To display an alert box with a note icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `NoteAlert` function.

```
FUNCTION NoteAlert (alertID: Integer;
                    filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `NoteAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `NoteAlert` draws the note icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The note icon has the following resource ID:

```
CONST noteIcon = 1;  {note icon}
```

By default, the Dialog Manager uses the standard note icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a note alert to inform users of a minor mistake that won't have any disastrous consequences if left as is. Usually this type of alert simply offers information, and the user responds by clicking an OK button. Occasionally, a note alert may ask a simple question and provide a choice of responses.

SEE ALSO

Figure 6-3 on page 6-8 illustrates the note icon in a typical note alert. Except that it includes a note icon in the alert box, `NoteAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

CautionAlert

To display an alert box with a caution icon in its upper-left corner (or, if appropriate for the alert stage, to play an alert sound instead of or in addition to displaying the alert box), use the `CautionAlert` function.

```
FUNCTION CautionAlert (alertID: Integer;
                       filterProc: ModalFilterProcPtr): Integer;
```

`alertID` The resource ID of an alert resource. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`filterProc`
A pointer to a function that responds to events not handled by the `ModalDialog` procedure. If you set this parameter to `NIL`, the Dialog Manager uses the standard event filter function, which allows users to press the Return or Enter key in lieu of clicking the default button. However, your application should provide a simple event filter function that also allows background applications to receive update events. Pass a pointer to the event filter function in this parameter.

DESCRIPTION

The `CautionAlert` function is the same as the `Alert` function except that, before drawing the items in the alert box, `CautionAlert` draws the caution icon in the upper-left corner (within the rectangle with local coordinates [10,20,42,52]). The caution icon has the following resource ID:

```
CONST cautionIcon = 2;  {caution icon}
```

By default, the Dialog Manager uses the standard caution icon from the System file. You can change this icon by providing your own `'ICON'` resource with this resource ID number.

Use a caution alert to alert the user of an operation that may have undesirable results if it's allowed to continue. Give the user the choice of continuing the action (by clicking an OK button) or stopping it (by clicking a Cancel button).

SEE ALSO

Figure 6-4 on page 6-9 illustrates the caution icon in a typical caution alert. Except that it includes a caution icon in the alert box, `CautionAlert` is identical to the `Alert` function. See the description of the `Alert` function on page 6-106 for detailed information about the parameters and behavior of both of these functions.

GetAlertStage

To determine the stage of the last occurrence of an alert, use the `GetAlertStage` function. The `GetAlertStage` function is also available as the `GetAlrtStage` function.

```
FUNCTION GetAlertStage: Integer;
```

DESCRIPTION

The `GetAlertStage` function returns a number from 0 to 3 as the stage of the last occurrence of an alert. For example, you can use the `GetAlertStage` function to ensure that your application deactivates the active window only if an alert box is to be displayed at that stage.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `ACount` contains this number. In addition, the global variable `ANumber` contains the resource ID of the alert resource of the last alert that occurred.

SEE ALSO

Listing 6-19 on page 6-66 illustrates how to use `GetAlertStage` to determine whether to deactivate a window for the current alert stage. Listing 6-2 on page 6-21 illustrates how to use an alert resource to specify different alert responses according to different alert stages.

ResetAlertStage

To reset the current alert stage to the first alert stage, use the `ResetAlertStage` procedure. The `ResetAlertStage` procedure is also available as the `ResetAlrtStage` procedure.

```
PROCEDURE ResetAlertStage;
```

DESCRIPTION

The `ResetAlertStage` procedure resets every alert to a first-stage alert.

SEE ALSO

Listing 6-2 on page 6-21 illustrates how to use an alert resource to specify different alert responses according to different alert stages.

Creating and Disposing of Dialog Boxes

To create a dialog box, you should generally use the `GetNewDialog` function, which takes information about the dialog from a dialog resource in a resource file. Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages. However, you can also use the `NewDialog` and `NewColorDialog` functions—for which you pass descriptive information in parameters—to create dialog boxes.

The `NewColorDialog` function is identical to the `NewDialog` function, except that `NewColorDialog` returns a pointer to a color graphics port.

When you no longer need a dialog box, use the `CloseDialog` procedure if you allocated the memory for the dialog record of the dialog box and use the `DisposeDialog` procedure if you did not. (To merely make the dialog box invisible to the user, you can use the Window Manager procedure `HideWindow`.)

GetNewDialog

To create a dialog box from a description in a dialog resource, use the `GetNewDialog` function.

```
FUNCTION GetNewDialog (dialogID: Integer; dStorage: Ptr;
                       behind: WindowPtr): DialogPtr;
```

`dialogID` The resource ID of a dialog resource. If the dialog resource is missing, the Dialog Manager returns to your application without creating the dialog box.

`dStorage` A pointer to the memory for the dialog record. If you set this parameter to `NIL` for modal dialog boxes and movable modal dialog boxes, the Dialog Manager automatically allocates memory for them in your application heap. For a modeless dialog box, however, you should allocate your own memory as you would for a window—otherwise, your heap could become fragmented.

`behind` A pointer to the window behind which the dialog box is to be placed on the desktop. Always set this parameter to the window pointer `Pointer(-1)` to bring the dialog box in front of all other windows.

DESCRIPTION

The `GetNewDialog` function creates a dialog record from the information in the dialog resource and returns a pointer to it. You can use this pointer with Window Manager or QuickDraw routines to manipulate the dialog box. If the dialog resource specifies that the dialog box should be visible, the dialog box is displayed. If the dialog resource specifies that the dialog box should initially be invisible, use the Window Manager procedure `ShowWindow` to display the dialog box.

If you supply a dialog color table resource with the same resource ID as the dialog resource, `GetNewDialog` uses the `NewColorDialog` function and returns a pointer to a color graphics port. If no dialog color table resource is present, `GetNewDialog` uses `NewDialog` to return a pointer to a black-and-white graphics port, although system software draws the window frame using the system's default colors.

The `dStorage` and `behind` parameters of `GetNewDialog` have the same meaning as they do in the Window Manager function `GetNewWindow`. Always set the `behind` parameter to `Pointer(-1)` to bring the dialog box to the front.

The dialog resource contains the resource ID of the dialog box's item list resource. After calling the Resource Manager to read the item list resource into memory (if it's not already in memory), `GetNewDialog` makes a copy of the item list resource and uses that copy; thus you may have several dialog boxes with identical items.

If you provide a dialog color table resource, `GetNewDialog` copies it before passing it to the Window Manager routine `SetWinColor` unless the number-of-entries element of the dialog color table resource is set to –1, in which case the default window colors are used instead. The `GetNewDialog` function makes the copy so that the dialog color table resource can be purged without affecting the dialog box.

SPECIAL CONSIDERATIONS

The `GetNewDialog` function doesn't release the memory occupied by the resources. Therefore, your application should mark all resources used for a dialog box as purgeable.

If either the dialog resource or the item list resource can't be read, the function result is `NIL`; your application should test to ensure that `NIL` is not returned before performing any more operations with the dialog box or its items.

For modal dialog boxes, the Dialog Manager function `ModalDialog` traps all events. This prevents your event loop from receiving activate events for your windows. Thus, if one of your application's windows is active when you use `GetNewDialog` to create a modal dialog box, you must explicitly deactivate that window before displaying the modal dialog box.

If you ever need to display a dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you do not use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to the dialog box that your application presents.

The `GetNewDialog` function uses either `NewDialog` or `NewColorDialog`, each of which generates an update event for the entire window contents. Thus, with the

exception of controls, items aren't drawn immediately. The Dialog Manager calls the Control Manager to draw controls, and the Control Manager draws them immediately. So the controls won't be drawn twice, the Dialog Manager calls the Window Manager procedure `ValidRect` for the enclosing rectangle of each control. If you find that there is too great a lag between the drawing of controls and the drawing of other items, try making the dialog box initially invisible and then calling the Window Manager procedure `ShowWindow` to show it.

SEE ALSO

See "Creating Dialog Boxes" beginning on page 6-23 and "Displaying Alert and Dialog Boxes" beginning on page 6-61 for discussions and examples of how to use `GetNewDialog`.

The `GetNewWindow` and `ShowWindow` procedures are described in the chapter "Window Manager" of this book. The Notification Manager is described in the chapter "Notification Manager" in *Inside Macintosh: Processes*.

"Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 discuss menu adjustment when your application displays dialog boxes. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about messages and control titles in dialog boxes.

NewColorDialog

To create a dialog box, you can use the `NewColorDialog` function, which returns a pointer to a color graphics port. Generally, you should instead use `GetNewDialog` to create a dialog box, because `GetNewDialog` takes information about the dialog box from a dialog resource in a resource file. (Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages.) The `NewColorDialog` function is also available as the `NewCDialog` function.

```
FUNCTION NewColorDialog (dStorage: Ptr; boundsRect: Rect;
                         title: Str255; visible: Boolean;
                         procID: Integer; behind: WindowPtr;
                         goAwayFlag: Boolean; refCon: LongInt;
                         items: Handle): CDialogPtr;
```

`dStorage` A pointer to the memory for the dialog record. If you set this parameter to `NIL` for modal dialog boxes and movable modal dialog boxes, the Dialog Manager allocates memory for them on your application heap. For a modeless dialog box, however, you should allocate your own memory as you would for a window—otherwise, your heap could become fragmented.

`boundsRect`
: A rectangle, given in global coordinates, that determines the size and position of the dialog box; these coordinates specify the upper-left and lower-right corners of the dialog box.

`title`
: A text string used for the title of a modeless or movable modal dialog box. You can specify an empty string (not `NIL`) for a title bar that contains no text.

`visible`
: A flag that specifies whether the dialog box should be drawn on the screen immediately. If you set this parameter to `FALSE`, the dialog box is not drawn until your application uses the Window Manager procedure `ShowWindow` to display it.

`procID`
: The window definition ID for the type of dialog box. Use the `dBoxProc` constant to specify modal dialog boxes, the `noGrowDocProc` constant to specify modeless dialog boxes, and the `movableDBoxProc` constant to specify movable modal dialog boxes.

`behind`
: A pointer to the window behind which the dialog box is to be placed on the desktop. Always set this parameter to the window pointer `Pointer(-1)` to bring the dialog box in front of all other windows.

`goAwayFlag`
: A flag to specify whether a modeless dialog box should have a close box in its title bar when the dialog box is active. If you set this parameter to `TRUE`, the dialog window has a close box in its title bar when the window is active; only modeless dialog boxes should have close boxes.

`refCon`
: A value that the Dialog Manager uses to set the `refCon` field of the dialog box's window record. Your application may store any value here for any purpose. For example, your application can store a number that represents a dialog box type, or it can store a handle to a record that maintains state information about the dialog box. You can use the Window Manager procedure `SetWRefCon` at any time to change this value in the dialog record for a dialog box, and you can use the `GetWRefCon` function to determine its current value.

`items`
: A handle to an item list resource for the dialog box. You can get the handle by calling the Resource Manager function `GetResource` to read the item list resource into memory. Use the Memory Manager procedure `HNoPurge` to make the handle unpurgeable while you use it or use the Operating System utility function `HandToHand` to make a copy of the handle and use the copy.

DESCRIPTION

The `NewColorDialog` function creates a dialog box as specified by its parameters and returns a pointer to a color graphics port for the new dialog box. The first eight parameters (`dStorage` through `refCon`) are passed to the Window Manager function `NewCWindow`, which creates the dialog box. You can use this pointer with Window Manager or QuickDraw routines to manipulate the dialog box.

The Dialog Manager uses the default window colors for the dialog box. By using the system's default colors, you ensure that your application's interface is consistent with

that of the Finder and other applications. However, if you absolutely feel compelled to break from this consistency, you can use the Window Manager procedure `SetWinColor` to use your own dialog color table resource that specifies colors other than the default colors. Be aware, however, that nonstandard colors in your alert and dialog boxes may initially confuse your users.

The Window Manager creates an auxiliary window record for the color dialog box. You can access this record with the Window Manager function `GetAuxWin`. (The `dialogCItemhandle` field of the auxiliary window record points to the dialog box's item color table resource.) If the dialog box's content color isn't white, it's a good idea to call `NewColorDialog` with the `visible` flag set to `FALSE`. After the color table and color item list resource are installed, use the Window Manager procedure `ShowWindow` to display the dialog box if it's the frontmost window. If the dialog box is a modeless dialog box that is not in front, use the Window Manager procedure `ShowHide` to display it.

When specifying the size and position of the dialog box in the `boundsRect` parameter, you should generally try to center dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is more appropriate. Also ensure that the tops of dialog boxes (including the title bars of modeless and movable modal dialog boxes) lie below the menu bar when you position them on the main screen. You can use the Menu Manager function `GetMBarHeight` to determine the height of the menu bar.

SPECIAL CONSIDERATIONS

For modal dialog boxes, the Dialog Manager function `ModalDialog` traps all events. This prevents your event loop from receiving activate events for your windows. Thus, if one of your application's windows is active when you use `NewColorDialog` to create a modal dialog box, you must explicitly deactivate that window before displaying the modal dialog box.

If you ever need to display a dialog box while your application is running in the background or is otherwise invisible to the user, you should use the Notification Manager to post a notification to the user. The Notification Manager automatically displays an alert box containing whatever message you specify; you do not need to use the Dialog Manager to create the alert box yourself.

Note that the Notification Manager provides a one-way communications path from your application to the user. There is no provision for carrying information back from the user to your application while it is in the background (although it is possible for your application to determine if the notification was received). If you need to solicit information from the user, use the Notification Manager to inform the user to bring your application to the foreground, where the user can then respond to the dialog box that your application presents.

The `NewColorDialog` function generates an update event for the entire window contents. Thus, with the exception of controls, items aren't drawn immediately. The Dialog Manager calls the Control Manager to draw controls, and the Control Manager draws them immediately. So that the controls won't be drawn twice, the Dialog Manager

calls the Window Manager procedure `ValidRect` for the enclosing rectangle of each control. If you find that there is too great a lag between the drawing of controls and the drawing of other items, try making the dialog box initially invisible and then calling the Window Manager procedure `ShowWindow` to show it.

SEE ALSO

Window Manager routines are described in the chapter "Window Manager" in this book. The Notification Manager is described in the chapter "Notification Manager" in *Inside Macintosh: Processes.* See *Inside Macintosh: Memory* for a description of `HNoPurge`. See *Inside Macintosh: Operating System Utilities* for a description of `HandToHand`.

"Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73 discuss menu bar adjustment when your application displays dialog boxes. See "Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 for recommendations about messages and control titles in dialog boxes. The `GetResource` function is described in the chapter "Resource Manager" of *Inside Macintosh: More Macintosh Toolbox*.

NewDialog

To create a dialog box, you can use the `NewDialog` function, which returns a pointer to a black-and-white graphics port (although system software draws the window frame of the dialog box using the system's default window colors). Generally, you should instead use `GetNewDialog` to create a dialog box; `GetNewDialog` takes information about the dialog from a dialog resource in a resource file. (Like window resources, dialog resources isolate descriptive information from your application code for ease of modification or translation to other languages.)

The `NewDialog` function is identical to the `NewColorDialog` function, except that `NewDialog` returns a pointer to a black-and-white graphics port. See the discussion of `NewColorDialog` on page 6-115 for descriptions of the parameters that you also pass to `NewDialog`.

```
FUNCTION NewDialog (dStorage: Ptr; boundsRect: Rect;
                    title: Str255; visible: Boolean;
                    procID: Integer; behind: WindowPtr;
                    goAwayFlag: Boolean; refCon: LongInt;
                    items: Handle):  DialogPtr;
```

DESCRIPTION

The `NewDialog` function creates a dialog box as specified by its parameters and returns a pointer to a black-and-white graphics port for the new dialog box. The first eight parameters (`dStorage` through `refCon`) are passed to the Window Manager function `NewWindow`, which creates the dialog box.

When specifying the size and position of the dialog box in the `boundsRect` parameter, you should generally try to center dialog boxes between the left and right margins of the screen or the window where the user is working, whichever is more appropriate. Also ensure that the tops of dialog boxes (including the title bars of modeless and movable modal dialog boxes) lie below the menu bar when you position them on the main screen. You can use the Menu Manager function `GetMBarHeight` to determine the height of the menu bar.

SEE ALSO

If you use a dialog color table resource to change the default window colors, use the `NewColorDialog` function, which returns a pointer to a color graphics port. See the description of `NewColorDialog` on page 6-115 for additional information common to both the `NewDialog` and `NewColorDialog` functions.

CloseDialog

To dismiss a dialog box for whose dialog record you allocated memory, use the `CloseDialog` procedure.

```
PROCEDURE CloseDialog (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `CloseDialog` procedure removes a dialog box from the screen and deletes it from the window list. The `CloseDialog` procedure releases the memory occupied by

- the data structures associated with the dialog box (such as its structure, content, and update regions)
- all the items in the dialog box (except for pictures and icons, which might be shared by other resources) and any data structures associated with them

Generally, you should provide memory for the dialog record of modeless dialog boxes when you create them. (You can let the Dialog Manager provide memory for modal and movable modal dialog boxes.) You should then use `CloseDialog` to close a modeless dialog box when the user clicks the close box or chooses Close from the File menu.

Because `CloseDialog` does not dispose of the dialog resource or the item list resource, it is important to make these resources purgeable. Unlike `GetNewDialog`, `NewColorDialog` does not use a copy of the item list resource. Thus, if you use `NewColorDialog` to create a dialog box, you may want to use `CloseDialog` to keep the item list resource in memory even if you didn't supply a pointer to the memory.

SEE ALSO

If you let the Dialog Manager allocate memory for the dialog box (by passing `NIL` in the `dStorage` parameter to the `GetNewDialog`, `NewColorDialog`, or `NewDialog` function), use the `DisposeDialog` procedure, described next, instead of `CloseDialog`.

DisposeDialog

To dismiss a dialog box for which the Dialog Manager supplies memory, use the `DisposeDialog` procedure. The `DisposeDialog` procedure is also available as the `DisposDialog` procedure.

```
PROCEDURE DisposeDialog (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DisposeDialog` procedure calls the `CloseDialog` procedure and, in addition, releases the memory occupied by the dialog box's item list resource and the dialog record. Call `DisposeDialog` when you're done with a dialog box if you pass `NIL` in the `dStorage` parameter to `GetNewDialog`, `NewColorDialog`, or `NewDialog`.

Generally, your application should not allocate memory for the dialog records of modal dialog boxes or movable modal dialog boxes. In these cases your application should use `DisposeDialog` when the user clicks the OK or Cancel button.

SEE ALSO

If you allocate memory for the dialog box (for example, by passing a pointer in the `dStorage` parameter to the `GetNewDialog`, `NewColorDialog`, or `NewDialog` function), use `CloseDialog`, described on page 6-119, instead of `DisposeDialog`.

Manipulating Items in Alert and Dialog Boxes

In many cases, you won't have to make any changes to alert or dialog boxes after you define them in the resource file. If you do need to make changes, use the Dialog Manager routines described in this section.

For most item manipulation, first call the `GetDialogItem` procedure to get the information about the item. You can then use other routines to manipulate that item. Use the `SetDialogItem` procedure if you use any of these other routines to change the item. You must also use `SetDialogItem` to install any of your own application-defined draw procedures. If you use `SetDialogItem`, make the dialog box initially invisible, change the item as appropriate, then make the dialog box visible by using the Window Manager procedure `ShowWindow`. (For information about manipulating text in an alert box or a dialog box, see "Handling Text in Alert and Dialog Boxes" beginning on page 6-129.)

You can dynamically add items to and remove items from a dialog box by using the `AppendDITL` and `ShortenDITL` procedures. These procedures are especially useful if you share a single item list resource among multiple dialog boxes, because you can then use `AppendDITL` or `ShortenDITL` to add or remove items as appropriate for individual dialog boxes. You typically make such dialog boxes invisible, use the `AppendDITL` and `ShortenDITL` procedures as appropriate, then make the dialog boxes visible by using the Window Manager procedure `ShowWindow`.

GetDialogItem

To get a handle to an item so that you can manipulate it (for example, to determine its current value, to change it, or to install a pointer to a draw procedure for an application-defined item), use the `GetDialogItem` procedure. The `GetDialogItem` procedure is also available as the `GetDItem` procedure.

```
PROCEDURE GetDialogItem (theDialog: DialogPtr; itemNo: Integer;
                         VAR itemType: Integer; VAR item: Handle;
                         VAR box: Rect);
```

`theDialog` A pointer to a dialog record.

`itemNo` A number corresponding to the position of an item in the dialog box's item list resource.

`itemType` A value that represents the type of item requested in the `itemNo` parameter. You can use any of these constants to determine the value returned in this parameter:

```
CONST
 ctrlItem    = 4;     {add this constant to the next }
                      { four constants}
 btnCtrl     = 0;     {standard button control}
 chkCtrl     = 1;     {standard checkbox control}
 radCtrl     = 2;     {standard radio button}
 resCtrl     = 3;     {control defined in a 'CNTL'}
 helpItem    = 1;     {help balloons}
 statText    = 8;     {static text}
 editText    = 16;    {editable text}
 iconItem    = 32;    {icon}
 picItem     = 64;    {QuickDraw picture}
 userItem    = 0;     {application-defined item}
 itemDisable = 128;   {add to any of the above to }
                      { disable it}
```

`item` For an application-defined draw procedure, a pointer to the draw procedure (coerced to a handle), returned for the item specified in the `itemNo` parameter; for all other item types, a handle to the item.

`box` The display rectangle (described in coordinates local to the dialog box), returned for the item specified in the `itemNo` parameter.

DESCRIPTION

The `GetDialogItem` procedure returns in its parameters the following information about the item numbered `itemNo` in the item list resource of the specified dialog box: in the `itemType` parameter, the item type; in the `item` parameter, a handle to the item (or, for application-defined draw procedures, the procedure pointer); and in the `box` parameter, the display rectangle for the item.

For most item manipulation, first use the `GetDialogItem` procedure to get the information about the item. You can then use other routines, such as `GetDialogItemText` and `SetDialogItem`, to determine and change the value of that item.

SEE ALSO

Listing 6-12 on page 6-49 illustrates the use of `GetDialogItem` in conjunction with `GetDialogItemText` to retrieve the text entered by a user in an editable text item. Listing 6-16 on page 6-58 illustrates the use of `GetDialogItem` in conjunction with `SetDialogItem` to install the draw procedure for an application-defined item into a dialog box. Listing 6-26 on page 6-83 illustrates the use of `GetDialogItem` to determine the current value of a checkbox in a dialog box.

SetDialogItem

After using the `GetDialogItem` procedure to get a handle to an item from a dialog box, use the `SetDialogItem` procedure to set or change the item. The `SetDialogItem` procedure is also available as the `SetDItem` procedure.

```
PROCEDURE SetDialogItem (theDialog: DialogPtr; itemNo: Integer;
                         itemType: Integer; item: Handle;
                         box: Rect);
```

`theDialog` A pointer to a dialog record.

`itemNo` A number corresponding to the position of an item in the dialog box's item list resource.

`itemType` A value that represents the type of item in the `itemNo` parameter. To specify the value for this parameter, you can use any of the constants listed on page 6-121 for the `itemType` parameter of the `GetDialogItem` procedure.

`item`	For an application-defined item, a pointer to the draw procedure (coerced to a handle) for the item specified in the `itemNo` parameter; for all other item types, a handle to the item.
`box`	The display rectangle (described in coordinates local to the dialog box) for the item specified in the `itemNo` parameter.

DESCRIPTION

The `SetDialogItem` procedure sets the item specified by the `itemNo` parameter for the specified dialog box. This procedure installs the item without drawing it; typically you create an invisible dialog box, use `SetDialogItem`, then use the Window Manager procedure `ShowWindow` to draw the dialog box and its items.

SEE ALSO

Listing 6-16 on page 6-58 illustrates how to use `SetDialogItem` to install an application-defined draw procedure. The `ShowWindow` procedure is described in the chapter "Window Manager" of this book.

HideDialogItem

Although you should rarely need to do so, you can make an item in a dialog box invisible by using the `HideDialogItem` procedure. The `HideDialogItem` procedure is also available as the `HideDItem` procedure.

```
PROCEDURE HideDialogItem (theDialog: DialogPtr; itemNo: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an item in the dialog box's item list resource.

DESCRIPTION

The `HideDialogItem` procedure hides the item specified by `itemNo` by giving it a display rectangle that's off the screen. Specifically, if the left coordinate of the item's display rectangle is less than 8192 (hexadecimal $2000), `HideDialogItem` adds 16,384 (hexadecimal $4000) to both the left and right coordinates of the rectangle. If the item is already hidden (that is, if the left coordinate is greater than 8192), `HideDialogItem` does nothing. To redisplay an item that's been hidden by `HideDialogItem`, you can use the `ShowDialogItem` procedure.

SPECIAL CONSIDERATIONS

If your application needs to display a number of dialog boxes that are similar except for one or two items, it's generally easier to modify the common elements using the `AppendDITL` and `ShortenDITL` procedures than to use the `HideDialogItem` and `ShowDialogItem` procedures.

The rectangle for a static text item must always be at least as wide as the first character of the text.

You generally shouldn't use `HideDialogItem` to make an editable text item invisible, because as the user presses the Tab key, the Dialog Manager attempts to move the cursor to the hidden editable text item, where the user's subsequent keystrokes will be placed.

ShowDialogItem

To redisplay an item that has been hidden by the `HideDialogItem` procedure, use the `ShowDialogItem` procedure. The `ShowDialogItem` procedure is also available as the `ShowDItem` procedure.

```
PROCEDURE ShowDialogItem (theDialog: DialogPtr; itemNo: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an item in the dialog box's item list resource.

DESCRIPTION

The `ShowDialogItem` procedure redisplays the item specified in `itemNo` by restoring the display rectangle the item had prior to the `HideDialogItem` call. Specifically, if the left coordinate of the item's display rectangle is greater than 8192, `ShowDialogItem` subtracts 16,384 from both the left and right coordinates of the rectangle. If the item is already visible (that is, if the left coordinate is less than 8192), `ShowDialogItem` does nothing.

The `ShowDialogItem` procedure adds the rectangle that contained the item to the update region so that it will be drawn. Note that if the item is a control you define in a control (`'CNTL'`) resource, the rectangle added to the update region is the rectangle defined in the control resource, not the display rectangle defined in the item list resource. If the item is an editable text item, `ShowDialogItem` activates it by calling the TextEdit procedure `TEActivate`.

FindDialogItem

To determine the item number of an item at a particular location in a dialog box, use the `FindDialogItem` function. The `FindDialogItem` function is also available as the `FindDItem` function.

```
FUNCTION FindDialogItem (theDialog: DialogPtr; thePt: Point)
                         : Integer;
```

`theDialog` A pointer to a dialog record.

`thePt` A point, specified in coordinates local to the dialog box.

DESCRIPTION

If the point specified in the parameter `thePt` lies within an item, `FindDialogItem` returns a number corresponding to the position of that item in the dialog box's item list resource. If the point doesn't lie within the item's rectangle, `FindDialogItem` returns –1. If items overlap, `FindDialogItem` returns the item number of the first item, in the item list resource, containing the point.

This function is useful for changing the cursor when it's over a particular item.

The `FindDialogItem` function returns 0 for the first item in the item list resource, 1 for the second, and so on. To get the proper item number before calling the `GetDialogItem` or `SetDialogItem` procedure, add 1 to `FindDialogItem`'s function result, as shown here:

```
theItem := FindDialogItem(theDialog, thePoint) + 1;
```

Note that `FindDialogItem` returns the item number of disabled items as well as enabled items.

AppendDITL

To add items to an existing dialog box while your application is running, use the `AppendDITL` procedure.

```
PROCEDURE AppendDITL (theDialog: DialogPtr; theDITL: Handle;
                      theMethod: DITLMethod);
```

`theDialog` A pointer to a dialog record. This is the dialog record to which you will add the item list resource specified in the parameter `theDITL`.

`theDITL` A handle to the item list resource whose items you want to append to the dialog box.

`theMethod` The manner in which you want the new items to be displayed in the existing dialog box. You can pass a negative value to offset the appended items from a particular item in the existing dialog box. You can also pass any of these constants:

```
CONST
overlayDITL =         0;    {overlay existing items}
appendDITLRight =     1;    {append at right}
appendDITLBottom =    2;    {append at bottom}
```

DESCRIPTION

The `AppendDITL` procedure adds the items in the item list resource specified in the parameter `theDITL` to the items of a dialog box. This procedure is especially useful if several dialog boxes share a single item list resource, because you can use `AppendDITL` to add items that are appropriate for individual dialog boxes. Your application can use the Resource Manager function `GetResource` to get a handle to the item list resource whose items you wish to add.

In the parameter `theMethod`, you specify how to append the new items, as follows:

- If you use the `overlayDITL` constant, `AppendDITL` superimposes the appended items over the dialog box. That is, `AppendDITL` interprets the coordinates of the display rectangles for the appended items (as specified in their item list resource) as local coordinates within the dialog box.
- If you use the `appendDITLRight` constant, `AppendDITL` appends the items to the right of the dialog box by positioning the display rectangles of the appended items relative to the upper-right coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.
- If you use the `appendDITLBottom` constant, `AppendDITL` appends the items to the bottom of the dialog box by positioning the display rectangles of the appended items relative to the lower-left coordinate of the dialog box. The `AppendDITL` procedure automatically expands the dialog box to accommodate the new dialog items.
- You can also append a list of items relative to an existing item by passing a negative number in the parameter `theMethod`. The absolute value of this number is interpreted as the item in the dialog box relative to which the new items are to be positioned. For example, if you pass –2, the display rectangles of the appended items are offset relative to the upper-left corner of item number 2 in the dialog box.

You typically create an invisible dialog box, call the `AppendDITL` procedure, then make the dialog box visible by using the Window Manager procedure `ShowWindow`.

SPECIAL CONSIDERATIONS

The `AppendDITL` procedure modifies the contents of the dialog box (for instance, by enlarging it). To use an unmodified version of the dialog box at a later time, your application should use the Resource Manager procedure `ReleaseResource` to release the memory occupied by the appended item list resource. Otherwise, if your application calls `AppendDITL` to add items to that dialog box again, the dialog box remains

modified by your previous call—for example, it will still be longer at the bottom if you previously used the `appendDITLBottom` constant.

The `AppendDITL` procedure is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `AppendDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `AppendDITL` is available.

SEE ALSO

Listing 6-13 on page 6-54 and Listing 6-14 on page 6-55 illustrate a typical use of `AppendDITL`. Figure 6-29 on page 6-52 shows the result of using the `overlayDITL` constant, Figure 6-30 on page 6-52 shows the result of using the `appendDITLRight` constant, Figure 6-31 on page 6-53 shows the result of using the `appendDITLBottom` constant, and Figure 6-32 on page 6-53 shows the result of using a negative number in the parameter `theMethod`.

The chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* describes the `GetResource` and `ReleaseResource` routines. The `Gestalt` function is described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities.* See the chapter "Window Manager" in this book for information about `ShowWindow`.

ShortenDITL

To remove items from an existing dialog box while your application is running, use the `ShortenDITL` procedure.

```
PROCEDURE ShortenDITL (theDialog: DialogPtr;
                       numberItems: Integer);
```

`theDialog` A pointer to a dialog record.

`numberItems`
The number of items to remove (starting from the last item in the item list resource).

DESCRIPTION

The `ShortenDITL` procedure removes the specified number of items from the dialog box. This procedure is especially useful if several dialog boxes share a single item list resource, because you can use `ShortenDITL` to remove items as necessary for individual dialog boxes.

You typically create an invisible dialog box, call the `ShortenDITL` procedure, then make the dialog box visible by using the Window Manager procedure `ShowWindow`. Note that `ShortenDITL` does not automatically resize the dialog box; you can use the Window Manager procedure `SizeWindow` if you need to resize the dialog box.

SPECIAL CONSIDERATIONS

The `ShortenDITL` procedure is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `ShortenDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `ShortenDITL` is available.

SEE ALSO

You can use the `CountDITL` function, described next, to determine the number of items in the dialog box's item list resource. See the chapter "Window Manager" in this book for information on the `ShowWindow` and `SizeWindow` procedures. The `Gestalt` function is described in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities.*

CountDITL

You can determine the number of items in a dialog box by using the `CountDITL` function.

```
FUNCTION CountDITL (theDialog: DialogPtr): Integer;
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `CountDITL` function returns the number of current items in a dialog box. You typically use `CountDITL` in conjunction with `ShortenDITL` to remove items from a dialog box.

SPECIAL CONSIDERATIONS

The `CountDITL` function is available in System 7 and in earlier versions of the Communications Toolbox. Before calling `CountDITL`, you should make sure that it is available by using the `Gestalt` function with the `gestaltDITLExtAttr` selector. Test the bit indicated by the `gestaltDITLExtPresent` constant in the `response` parameter. If the bit is set, then `CountDITL` is available.

SEE ALSO

The `Gestalt` function is described in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities.*

Handling Text in Alert and Dialog Boxes

The Dialog Manager provides several routines for manipulating text. You can use the `ParamText` procedure to supply text strings, such as document titles, dynamically in the static text items of alert and dialog boxes. The `GetDialogItemText` and `SetDialogItemText` procedures are useful for determining and changing text in both static text and editable text items. You can use the `SelectDialogItemText` procedure to select and highlight text in an editable text item.

When a dialog box containing an editable text item is active, use the `DialogCut` procedure to handle the Cut editing command, the `DialogCopy` procedure to handle the Copy command, the `DialogPaste` procedure to handle the Paste command, and the `DialogDelete` procedure to handle the Clear command.

Once you determine that an event occurs in a modeless or movable modal dialog box, you can use the `DialogSelect` function, which is described on page 6-139, to handle key-down events in editable text items automatically. The `ModalDialog` procedure uses `DialogSelect` to handle key-down events in the editable text items of modal dialog boxes.

ParamText

To substitute text strings in the static text items of your alert or dialog boxes while your application is running, use the `ParamText` procedure.

```
PROCEDURE ParamText (param0: Str255; param1: Str255;
                     param2: Str255; param3: Str255);
```

`param0` A text string to substitute for the special string `^0` in the static text items of all subsequently created alert and dialog boxes.

`param1` A text string to substitute for the special string `^1` in the static text items of all subsequently created alert and dialog boxes.

`param2` A text string to substitute for the special string `^2` in the static text items of all subsequently created alert and dialog boxes.

`param3` A text string to substitute for the special string `^3` in the static text items of all subsequently created alert and dialog boxes.

DESCRIPTION

The `ParamText` procedure replaces the special strings `^0` through `^3` in the static text items of all subsequently created alert and dialog boxes with the text strings you pass as parameters. Pass empty strings (not `NIL`) for parameters not used.

SPECIAL CONSIDERATIONS

The strings used in `ParamText` are stored in the low-memory global variable `DAStrings`, which specifies a set of string handles used by the Dialog Manager.

If the user launches a desk accessory in your application's partition and the desk accessory calls `ParamText`, it may change the text in your application's dialog box.

You should be very careful about using `ParamText` in modeless dialog boxes. If a modeless dialog box using `ParamText` is onscreen and you display another dialog box or alert box that also uses `ParamText`, both boxes will be affected by the latest call to `ParamText`.

The strings you pass in the parameters to `ParamText` cannot contain the special strings `^0` through `^3`, or else the procedure will enter an endless loop of substitutions in versions of system software earlier than 7.1.

Note that you should try to store text strings in resource files to facilitate translation into other languages; therefore, `ParamText` is best used for supplying text strings, such as document names, that the user specifies. To avoid problems with grammar and sentence structure when you localize your application, you should use `ParamText` to supply only one text string per screen message.

SEE ALSO

Listing 6-9 on page 6-47 and Listing 6-10 on page 6-48 show an example of how you can use `ParamText` to supply the title of the user's current document to your alert and dialog boxes. If you need to supply a default text string to an editable text item while your application is running, use `SetDialogItemText`. The `SetDialogItemText` procedure also allows you to set or change the entire text string for a static text item.

GetDialogItemText

After using the `GetDialogItem` procedure to get a handle to an editable text item or a static text item in a dialog box, you can use the `GetDialogItemText` procedure to get the text string contained in that item. The `GetDialogItemText` procedure is also available as the `GetIText` procedure.

```
PROCEDURE GetDialogItemText (item: Handle; VAR text: Str255);
```

`item`	A handle to an editable text item or a static text item in a dialog box.
`text`	The text contained within the item.

DESCRIPTION

The `GetDialogItemText` procedure returns, in the `text` parameter, the text of the given editable text or static text item.

SPECIAL CONSIDERATIONS

If the user types more than 255 characters in an editable text item, `GetDialogItemText` returns only the first 255.

SEE ALSO

Listing 6-12 on page 6-49 illustrates how to use `GetDialogItemText` to retrieve the text that a user types into an editable text item.

SetDialogItemText

After using the `GetDialogItem` procedure to get a handle to an editable text item or a static text item in a dialog box, you can use the `SetDialogItemText` procedure to display a particular text string in that item. The `SetDialogItemText` procedure is also available as the `SetIText` procedure.

```
PROCEDURE SetDialogItemText (item: Handle; text: Str255);
```

`item`	A handle to an editable text item or a static text item in a dialog box.
`text`	The text to display in the item.

DESCRIPTION

The `SetDialogItemText` procedure places the specified text in the specified item and draws the item. This procedure is useful for supplying a default text string—such as a document name—for an editable text item while your application is running.

SPECIAL CONSIDERATIONS

All strings should be stored in resource files to ease translation into other languages.

SEE ALSO

For static text items, the `ParamText` procedure, described on page 6-129, is useful when you need to determine and provide only a portion of a text string while your application is running.

SelectDialogItemText

To select and highlight text contained in an editable text item, use the `SelectDialogItemText` procedure. The `SelectDialogItemText` procedure is also available as the `SelIText` procedure.

```
PROCEDURE SelectDialogItemText (theDialog: DialogPtr;
                                itemNo: Integer;
                                strtSel: Integer;
                                endSel: Integer);
```

`theDialog`	A pointer to a dialog record.
`itemNo`	A number corresponding to the position of an editable text item in the dialog box's item list resource.
`strtSel`	A number representing the position of the first character to begin selecting.
`endSel`	A number representing one position past the last character to be selected.

DESCRIPTION

If the item in the `itemNo` parameter is an editable text item that contains text, the `SelectDialogItemText` procedure sets the text selection range to extend from the character position specified in the `strtSel` parameter up to but not including the character position specified in the `endSel` parameter. The selection range is highlighted unless `strtSel` equals `endSel`, in which case a blinking vertical bar is displayed to indicate an insertion point at that position. If the editable text item doesn't contain text, `SelectDialogItemText` displays the insertion point.

You can select the entire text by specifying the number 0 in the `strtSel` parameter and the number 32767 in the `endSel` parameter.

For example, if the user makes an unacceptable entry in the editable text item, your application can display an alert box reporting the problem and then use `SelectDialogItemText` to select the entire text 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.

SEE ALSO

For details about text selection range and character position, see the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogCut

When a dialog box containing an editable text item is active, use the `DialogCut` procedure to handle the Cut editing command. The `DialogCut` procedure is also available as the `DlgCut` procedure.

```
PROCEDURE DialogCut (theDialog: DialogPtr);
```

`theDialog`	A pointer to a dialog record.

DESCRIPTION

The `DialogCut` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TECut` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TECut` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogCopy

When a dialog box containing an editable text item is active, use the `DialogCopy` procedure to handle the Copy editing command. The `DialogCopy` procedure is also available as the `DlgCopy` procedure.

```
PROCEDURE DialogCopy (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogCopy` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TECopy` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TECopy` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

DialogPaste

When a dialog box containing an editable text item is active, use the `DialogPaste` procedure to handle the Paste editing command. The `DialogPaste` procedure is also available as the `DlgPaste` procedure.

```
PROCEDURE DialogPaste (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogPaste` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TEPaste` to the selected editable text item. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TEPaste` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text.*

DialogDelete

When a dialog box containing an editable text item is active, use the `DialogDelete` procedure to handle the Clear editing command. The `DialogDelete` procedure is also available as the `DlgDelete` procedure.

```
PROCEDURE DialogDelete (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DialogDelete` procedure checks whether the dialog box has any editable text items and, if so, applies the TextEdit procedure `TEDelete` to the selected text. Your application should test whether a dialog box is the frontmost window when handling mouse-down events in the Edit menu and then call this routine when appropriate.

SEE ALSO

For more information about allowing access to your menus when your application displays dialog boxes, see "Adjusting Menus for Modal Dialog Boxes" beginning on page 6-68 and "Adjusting Menus for Movable Modal and Modeless Dialog Boxes" on page 6-73. The `TEDelete` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text*.

Handling Events in Dialog Boxes

Handling events in an alert box is very simple: after you invoke an alert box, the Dialog Manager handles most events for you by automatically calling the `ModalDialog` procedure. To handle events in a modal dialog box, your application must explicitly call the `ModalDialog` procedure after displaying the dialog box. In either case, when an enabled item is clicked, the Dialog Manager returns the item number. You'll then do whatever is appropriate in response to that click. For both alert and modal dialog boxes, you should also provide a simple event filter function that allows other windows to respond to update events and that allows your alert or dialog box to respond to a few key-down events for keys such as Return, Enter, and Esc.

You can use your normal event-handling code to determine whether an event occurs in a modeless or movable modal dialog box, or you can use the `IsDialogEvent` function to learn whether they need to be handled as part of a dialog box. Once you determine that an event occurs in a modeless or movable modal dialog box, you can use the `DialogSelect` function to handle key-down events in editable text items automatically, to handle update and activate events automatically, and to report the enabled items clicked by the user. You then respond as appropriate to clicks in your active items. Or you can use Control Manager, TextEdit, and Window Manager routines (such as `FindWindow`, `BeginUpdate`, `EndUpdate`, `FindControl`, `TrackControl`, and `TEClick`) to handle these events without the aid of the Dialog Manager.

ModalDialog

To handle events when you display a modal dialog box, use the `ModalDialog` procedure.

```
PROCEDURE ModalDialog (filterProc: ModalFilterProcPtr;
                       VAR itemHit: Integer);
```

`filterProc`
A pointer to an event filter function.

`itemHit` A number representing the position of the selected item in the item list resource for the active modal dialog box.

DESCRIPTION

Call the `ModalDialog` procedure immediately after displaying a modal dialog box. The `ModalDialog` procedure assumes that a modal dialog box is displayed as the current port, and `ModalDialog` repeatedly handles events inside that port until an event involving an enabled dialog box item—such as a click in a radio button, for example—occurs. If the event is a mouse-down event outside the content region of the dialog box, `ModalDialog` emits the system alert sound and gets the next event. After receiving an event involving an enabled item, `ModalDialog` returns its item number in the `itemHit` parameter. Your application should then do whatever is appropriate in response to an event in that item. Your application should continue calling `ModalDialog` until the user selects the OK or Cancel button.

For events inside the dialog box, `ModalDialog` passes the event to the event filter function pointed to in the `filterProc` parameter before handling the event. When the event filter returns `FALSE`, `ModalDialog` handles the event. If the event filter function handles the event, the event filter function returns `TRUE`, and `ModalDialog` performs no more event handling.

If you set the `filterProc` parameter to `NIL`, the standard event filter function is executed. The standard event filter function returns `TRUE` and causes `ModalDialog` to return item number 1, which is the number of the default button, when the user presses the Return key or the Enter key. However, your application should provide a simple event filter function that

- returns `TRUE` and the item number for the default button if the user presses the Return or Enter key
- returns `TRUE` and the item number for the Cancel button if the user presses the Esc key or the Command-period key combination
- updates your windows in response to update events (this allows background applications to receive update events) and return `FALSE`
- returns `FALSE` for all events that your event filter function doesn't handle

You can use the same event filter function in most or all of your alert and modal dialog boxes.

You can also use the event filter function specified in the `filterProc` parameter to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor within an application-defined item.

To handle events, `ModalDialog` calls the `IsDialogEvent` function. If the result of `IsDialogEvent` is `TRUE`, then `ModalDialog` calls the `DialogSelect` function to handle the event. Unless the event filter function returns `TRUE`, `ModalDialog` handles the event as follows:

- In response to an activate or update event for the dialog box, `ModalDialog` activates or updates its window.
- If the user presses the mouse button while the cursor is in an editable text item, `ModalDialog` responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If a key-down event occurs and there's an editable text item, `ModalDialog` uses TextEdit to handle text entry and editing automatically. If the editable text item is enabled, `ModalDialog` returns its item number after it receives either the mouse-down or key-down event. Normally, editable text items are disabled, and you use the `GetDialogItemText` procedure to read the information in the items only after the user clicks the OK button.
- If the user presses the mouse button while the cursor is in a control, `ModalDialog` calls the Control Manager function `TrackControl`. If the user releases the mouse button while the cursor is in an enabled control, `ModalDialog` returns the control's item number. Your application should respond appropriately—for example, by performing a command after the user clicks the OK button.

- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `ModalDialog` returns the item's number, and your application should respond appropriately. Generally, only controls should be enabled. If your application creates a control more complex than a button, radio button, or checkbox, your application must handle events inside that item with your event filter function.
- If the user presses the mouse button while the cursor is in a disabled item or in no item, or if any other event occurs, `ModalDialog` does nothing.

SPECIAL CONSIDERATIONS

Do not use `ModalDialog` for movable modal dialog boxes (that is, those created with the `movableDBoxProc` window definition ID) or for modeless dialog boxes (that is, those created with the `noGrowDocProc` window definition ID). If you want the Dialog Manager to assist you in handling events for movable modal and modeless dialog boxes, use the `IsDialogEvent` and `DialogSelect` functions instead.

The `ModalDialog` procedure calls the Event Manager function `GetNextEvent` with a mask that excludes disk-inserted events. To receive disk-inserted events, your event filter function can call the Event Manager procedure `SetSystemEventMask`.

When `ModalDialog` calls `TrackControl`, it does not allow you to specify the action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control, a picture, or an application-defined item that draws a control- like object in your dialog box. You must then provide an event filter function that appropriately handles events in that item.

SEE ALSO

Listing 6-26 on page 6-83 illustrates the use of `ModalDialog`. "Responding to Events in Editable Text Items" beginning on page 6-79 describes how `ModalDialog` uses TextEdit to handle text entry and editing in editable text items. The `IsDialogEvent` and `DialogSelect` functions (which your application may use instead of `ModalDialog` for modeless and movable modal dialog boxes) are described on page 6-138 and page 6-139, respectively. See the description of `MyEventFilter` on page 6-145 for information about the event filter function your application should specify in the `filterProc` parameter.

The `GetNextEvent` and `SetSystemEventMask` routines are described in the chapter "Event Manager" in this book. See that chapter as well for a discussion of disk-inserted events. See "Responding to Events in Controls" on page 6-78 for a description of how your application should respond to events inside of controls; the `TrackControl` function is fully described in the chapter "Control Manager" in this book. Also see that chapter for information about creating your own nonstandard controls. TextEdit is described in the chapter "TextEdit" of *Inside Macintosh: Text.*

IsDialogEvent

To determine whether a modeless dialog box or a movable modal dialog box is active when an event occurs, you can use the `IsDialogEvent` function.

```
FUNCTION IsDialogEvent (theEvent: EventRecord): Boolean;
```

`theEvent` An event record returned by an Event Manager function such as `WaitNextEvent`.

DESCRIPTION

If any event, including a null event, occurs when your dialog box is active, `IsDialogEvent` returns `TRUE`; otherwise, it returns `FALSE`. When `IsDialogEvent` returns `FALSE`, pass the event to the rest of your event-handling code. When `IsDialogEvent` returns `TRUE`, pass the event to `DialogSelect` after testing for the events that `DialogSelect` does not handle.

A dialog record includes a window record. When you use the `GetNewDialog`, `NewDialog`, or `NewColorDialog` function to create a dialog box, the Dialog Manager sets the `windowKind` field in the window record to `dialogKind`. To determine whether the active window is a dialog box, `IsDialogEvent` checks the `windowKind` field.

Before passing the event to `DialogSelect`, you should perform the following tests whenever `IsDialogEvent` returns `TRUE`:

- Check whether the event is a key-down event for the Return, Enter, Esc, or Command-period keystrokes. When the user presses the Return or Enter key, your application should respond as if the user had clicked the default button; when the user presses Esc or Command-period, your application should respond as if the user had clicked the Cancel button. Use the Control Manager procedure `HiliteControl` to highlight the applicable button for 8 ticks.
- At this point, you may also want to check for and respond to any special events that you do not wish to pass to `DialogSelect` or that require special processing before you pass them to `DialogSelect`. You would need to do this, for example, if the dialog box needs to respond to disk-inserted events.
- Check whether the event is an update event for a window other than the dialog box and, if it is, update your window.
- For complex items that you create, such as pictures or application-defined items that emulate complex controls, test for and respond to mouse events inside those items as appropriate. When `DialogSelect` calls `TrackControl`, it does not allow you to specify the action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control or a picture or an application-defined item that draws a control-like object in your dialog box. You must then test for and respond to those events yourself.

If your application uses `IsDialogEvent` to help handle events when you display a movable modal dialog box, perform the following additional tests before passing events to `DialogSelect`:

- Test for mouse-down events in the title bar of the movable modal dialog box and respond by dragging the dialog box accordingly.
- Test for and respond to mouse-down events in the Apple menu and, if the movable modal dialog box includes editable text items, in the Edit menu. (You should disable all other menus when you display a movable modal dialog box.)
- Play the system alert sound for every other mouse-down event outside the movable modal dialog box.

SPECIAL CONSIDERATIONS

Both `IsDialogEvent` and `DialogSelect` are unreliable when running in versions of system software earlier than System 7. You shouldn't use these routines if you expect your application to run in earlier versions of system software.

SEE ALSO

The `WaitNextEvent` function is described in the chapter "Event Manager" in this book. See *Inside Macintosh: Sound* for a description of the `SysBeep` procedure. The `FrontWindow` function is described in the chapter "Window Manager" in this book.

DialogSelect

After determining that an event related to an active modeless dialog box or an active movable modal dialog box has occurred, you can use the `DialogSelect` function to handle most of the events inside the dialog box.

```
FUNCTION DialogSelect (theEvent: EventRecord;
                       VAR theDialog: DialogPtr;
                       VAR itemHit: Integer): Boolean;
```

`theEvent`	An event record returned by an Event Manager function such as `WaitNextEvent`.
`theDialog`	A pointer to a dialog record for the dialog box where the event occurred.
`itemHit`	A number corresponding to the position of an item within the item list resource of the active dialog box.

DESCRIPTION

The `DialogSelect` function handles most of the events relating to a dialog box. If the event is an activate or update event for a dialog box, `DialogSelect` activates or updates it and returns `FALSE`. If the event involves an enabled item, `DialogSelect`

returns a function result of TRUE. In its itemHit parameter, it returns the item number of the item selected by the user. In the parameter theDialog, it returns a pointer to the dialog record for the dialog box where the event occurred. In all other cases, the DialogSelect function returns FALSE. When DialogSelect returns TRUE, do whatever is appropriate as a response to the event involving that item in that particular dialog box; when it returns FALSE, do nothing.

Generally, only controls should be enabled in a dialog box; therefore your application should normally respond only when DialogSelect returns TRUE after the user clicks an enabled control, such as the OK button.

The DialogSelect function first obtains a pointer to the window containing the event. For update and activate events, the event record contains the window pointer. For other types of events, DialogSelect calls the Window Manager function FrontWindow. The Dialog Manager then makes this window the current graphics port by calling the QuickDraw procedure SetPort. Then DialogSelect prepares to handle the event by setting up text information if there are any editable text items in the active dialog box.

If the event is an update event for a dialog box, DialogSelect calls the Window Manager procedure BeginUpdate, the Dialog Manager procedure DrawDialog, and then the Window Manager procedure EndUpdate. When an item is a control defined in a control ('CNTL') resource, the rectangle added to the update region is the rectangle defined in the control resource, not the display rectangle defined in the item list resource.

The DialogSelect function handles the event as follows:

- In response to an activate or update event for the dialog box, DialogSelect activates or updates its window and returns FALSE.
- If a key-down event or an auto-key event occurs and there's an editable text item in the dialog box, DialogSelect uses TextEdit to handle text entry and editing, and DialogSelect returns TRUE for a function result. In its itemHit parameter, DialogSelect returns the item number.
- If a key-down event or an auto-key event occurs and there's no editable text item in the dialog box, DialogSelect returns FALSE.
- If the user presses the mouse button while the cursor is in an editable text item, DialogSelect responds to the mouse activity as appropriate—that is, either by displaying an insertion point or by selecting text. If the editable text item is disabled, DialogSelect returns FALSE. If the editable text item is enabled, DialogSelect returns TRUE and in its itemHit parameter returns the item number. Normally, editable text items are disabled, and you use the GetDialogItemText function to read the information in the items only after the OK button is clicked.
- If the user presses the mouse button while the cursor is in a control, DialogSelect calls the Control Manager function TrackControl. If the user releases the mouse button while the cursor is in an enabled control, DialogSelect returns TRUE for a function result and in its itemHit parameter returns the control's item number. Your application should respond appropriately—for example, by performing a command after the user clicks the OK button.

- If the user presses the mouse button while the cursor is in any other enabled item in the dialog box, `DialogSelect` returns `TRUE` for a function result and in its `itemHit` parameter returns the item's number. Generally, only controls should be enabled. If your application creates a complex control—such as one that measures how far a dial is moved—your application must handle mouse events in that item before passing the event to `DialogSelect`.
- If the user presses the mouse button while the cursor is in a disabled item, or if it is in no item, or if any other event occurs, `DialogSelect` does nothing.
- If the event isn't one that `DialogSelect` specifically checks for (if it's a null event, for example), and if there's an editable text item in the dialog box, `DialogSelect` calls the TextEdit procedure `TEIdle` to make the insertion point blink.

SPECIAL CONSIDERATIONS

Because `DialogSelect` handles only mouse-down events in a dialog box and key-down events in a dialog box's editable text items, you should handle other events as appropriate before passing them to `DialogSelect`. Likewise, when `DialogSelect` calls `TrackControl`, it does not allow you to specify any action procedure necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control (for example, one that measures how long the user holds down the mouse button or how far the user has moved an indicator), you can create your own control or a picture or an application-defined item that draws a control-like object in your dialog box. You must then test for and respond to those events yourself.

Within dialog boxes, use the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support Cut, Copy, Paste, and Clear commands in editable text boxes.

The `DialogSelect` function is unreliable when running in versions of system software earlier than System 7. You shouldn't use this routine if you expect your application to run under earlier versions of system software.

SEE ALSO

Listing 6-25 on page 6-79 illustrates the use of `DialogSelect` to make the cursor blink in editable text items during null events; Listing 6-29 on page 6-92 illustrates the use of `DialogSelect` to handle mouse events in a modeless dialog box; Listing 6-33 on page 6-96 illustrates the use of `DialogSelect` to handle key-down events in editable text items; Listing 6-34 on page 6-98 illustrates the use of `DialogSelect` to handle activate events in a modeless dialog box.

DrawDialog

If you don't use any other Dialog Manager routines for handling events in a dialog box, you can use the `DrawDialog` procedure to draw its entire contents.

```
PROCEDURE DrawDialog (theDialog: DialogPtr);
```

`theDialog` A pointer to a dialog record.

DESCRIPTION

The `DrawDialog` procedure draws the entire contents of the specified dialog box. The `DrawDialog` procedure draws all dialog items, calls the Control Manager procedure `DrawControls` to draw all controls, and calls the TextEdit procedure `TEUpdate` to update all static and editable text items and to draw their display rectangles. The `DrawDialog` procedure also calls the application-defined items' draw procedures if the items' rectangles are within the update region.

The `DialogSelect`, `ModalDialog`, `Alert`, `StopAlert`, `NoteAlert`, and `CautionAlert` routines use `DrawDialog` automatically. If you use `GetNewDialog` to create a dialog box but don't use any of these other Dialog Manager routines when handling events in the dialog box, you can use `DrawDialog` to redraw the contents of the dialog box when it's visible. If the dialog box is invisible, first use the Window Manager procedure `ShowWindow` and then use `DrawDialog`.

SEE ALSO

See the chapters "Window Manager" and "Event Manager" in this book for more information on update and activate events for windows. The `DrawControls` procedure is described in the chapter "Control Manager" in this book. The `TEUpdate` procedure is described in the chapter "TextEdit" in *Inside Macintosh: Text.*

UpdateDialog

You can use the `UpdateDialog` procedure to redraw the update region of a specified dialog box. The `UpdateDialog` procedure is also available as the `UpdtDialog` procedure.

```
PROCEDURE UpdateDialog (theDialog: DialogPtr;
                        updateRgn: RgnHandle);
```

`theDialog` A pointer to a dialog record.

`updateRgn` A handle to the window region that needs to be updated.

DESCRIPTION

The `UpdateDialog` procedure redraws only the region in a dialog box specified in the `updateRgn` parameter. Because the `DialogSelect`, `ModalDialog`, `Alert`, `StopAlert`, `NoteAlert`, and `CautionAlert` routines automatically call `DrawDialog` to handle update events in your alert and dialog boxes, your application might never need to use `UpdateDialog`.

Instead of drawing the entire contents of the specified dialog box, `UpdateDialog` draws only the items in the specified update region. You can use `UpdateDialog` in response to an update event, and you should usually bracket it by calls to the Window Manager procedures `BeginUpdate` and `EndUpdate`. The `UpdateDialog` procedure uses the QuickDraw procedure `SetPort` to make the dialog box the current graphics port. For drawing controls, `UpdateDialog` uses the Control Manager procedure `UpdateControls`, which is faster than the `DrawControls` procedure.

SEE ALSO

Listing 6-35 on page 6-99 illustrates the use of `UpdateDialog` to respond to update events in a modeless dialog box. See the chapter "Window Manager" in this book for more information on update and activate events for windows. The `UpdateControls` procedure is described in the chapter "Control Manager" in this book.

Application-Defined Routines

If you supply an application-defined item in a dialog box, you must provide a draw procedure for the Dialog Manager to use when displaying the item; that procedure is referred to in this section as `MyItem`. If you want the Dialog Manager to play sounds other than the system alert sound, you must provide your own sound procedure, referred to in this section as `MyAlertSound`. To supplement the Dialog Manager's ability to handle events in the Macintosh multitasking environment, you should provide an event filter function that the Dialog Manager calls whenever it displays alert boxes and modal dialog boxes. This function is referred to as `MyEventFilter`.

MyItem

To draw your own application-defined item in a dialog box, provide a draw procedure that takes two parameters: a window pointer to the dialog box and an item number from the dialog box's item list resource. For example, this is how you should declare the procedure if you were to name it `MyItem`:

```
PROCEDURE MyItem (theWindow: WindowPtr; itemNo: Integer);
```

`theWindow` A pointer to the dialog record for the dialog box containing an application-defined item. If your procedure can draw in more than one dialog box, this parameter tells your procedure which one to draw in.

`itemNo` A number corresponding to the position of an item in the item list resource for the specified dialog box. If your procedure draws more than one item, this parameter tells your procedure which one to draw.

DESCRIPTION

The Dialog Manager calls your procedure to draw an application-defined item at the time you display the specified dialog box. When calling your draw procedure, the Dialog Manager sets the current port to the dialog box's graphics port. Normally, you create an invisible dialog box and then use the Window Manager procedure `ShowWindow` to display the dialog box.

Before you display the dialog box, use the `SetDialogItem` procedure to install this procedure in the dialog record. Before using `SetDialogItem`, you must first use the `GetDialogItem` procedure to obtain a handle to an item of type `userItem`.

If you enable the application-defined item that you draw with this procedure, the `ModalDialog` procedure and the `DialogSelect` function return the item's number when the user clicks that item. If your application needs to respond to a user action more complex than this (for example, if your application needs to measure how long the user holds down the mouse or how far the user drags the cursor), your application must track the cursor itself. If you use `ModalDialog`, your event filter function must handle events inside the item; if you use `DialogSelect`, your application must handle events inside the item before handing events to `DialogSelect`.

SEE ALSO

Listing 6-17 on page 6-59 illustrates a procedure that draws a bold outline around a button of any size and shape; Listing 6-16 on page 6-58 shows the use of `GetDialogItem` and `SetDialogItem` to install this draw procedure in a dialog record. The `ShowWindow` procedure is described in the chapter "Window Manager" in this book.

MyAlertSound

If you want the Dialog Manager to play sounds other than the system alert sound, write your own sound procedure and call the `ErrorSound` procedure to make it the current sound procedure. For example, you can declare a sound procedure named `MyAlertSound`, as shown here:

```
PROCEDURE MyAlertSound (soundNo: Integer);
```

`soundNo` An integer from 0 to 3, representing the four possible alert stages.

DESCRIPTION

For each of the four alert stages that can be reported in the `soundNo` parameter, your procedure can emit any sound that you define. When the Dialog Manager calls your procedure, it passes 0 as the sound number for alert sounds specified by the `silent` constant in the alert resource. The Dialog Manager passes 1 for sounds specified by the `sound1` constant, 2 for sounds specified by the `sound2` constant, and 3 for sounds specified by the `sound3` constant.

SPECIAL CONSIDERATIONS

When the Dialog Manager detects a click outside an alert box or a modal dialog box, it uses the Sound Manager procedure `SysBeep` to play the system alert sound. By changing settings in the Sound control panel, the user can select which sound to play as the system alert sound. For consistency with system software and other Macintosh applications, your sound procedure should call `SysBeep` whenever your sound procedure receives sound number 1 (which you can represent with the `sound1` constant).

SEE ALSO

Listing 6-3 on page 6-22 illustrates how to use `MyAlertSound`. The `SysBeep` procedure is described in *Inside Macintosh: Sound.*

MyEventFilter

To supplement the Dialog Manager's ability to handle events, your application should provide an event filter function that the Dialog Manager calls when it displays alert boxes and modal dialog boxes. Your event filter function should have three parameters and return a Boolean value. For example, this is how you would declare it if you were to name it `MyEventFilter`:

```
FUNCTION MyEventFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

`theDialog`	A pointer to a dialog record for an alert box or a modal dialog box.
`theEvent`	An event record returned by an Event Manager function such as `WaitNextEvent`.
`itemHit`	A number corresponding to the position of an item in the item list resource for the alert or modal dialog box.

DESCRIPTION

After receiving an event that it does not handle, your function should return `FALSE`. When your function returns `FALSE`, `ModalDialog` handles the event, which you pass in the parameter `theEvent`. (Your function can also change the event to simulate a different event and return `FALSE`, which passes the event to the Dialog Manager for handling.) If your function *does* handle the event, your function should return `TRUE` as a function result, and in the `itemHit` parameter return the number of the item that it handled. The `ModalDialog` procedure and, in turn, the `Alert`, `NoteAlert`, `StopAlert`, and `CautionAlert` functions then return this item number in their own `itemHit` parameters.

Your event filter function should perform the following tasks:

- return `TRUE` and the item number for the default button if the user presses Return or Enter
- return `TRUE` and the item number for the Cancel button if the user presses Esc or Command-period
- update your windows in response to update events (this allows background applications to receive update events) and return `FALSE`
- return `FALSE` for all events that your event filter function doesn't handle

You can also use the event filter function to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor in an application-defined item. For example, if you provide an application-defined item that requires you to measure how long the user holds down the mouse button or how far the user drags the cursor, use the event filter function to handle events inside that item.

The `ModalDialog` procedure calls the Event Manager function `GetNextEvent` with a mask that excludes disk-inserted events; to receive disk-inserted events, your event filter function can call the Event Manager procedure `SetSystemEventMask`.

You can use the same event filter function in most or all of your alert and modal dialog boxes.

For alert and modal dialog boxes, the Dialog Manager provides a standard event filter function that checks whether the user has pressed the Enter or Return key and, if so, returns the item number of the default button. Your event filter function should always check whether the Return key or Enter key was pressed and, if so, return the number of the default button in the `itemHit` parameter and a function result of `TRUE`.

In all alert and dialog boxes, any buttons that are activated by key sequences should invert to indicate which item has been selected. Use the Control Manager procedure `HiliteControl` to invert a button for 8 ticks, long enough to be noticeable but not so long as to be annoying. The Control Manager performs this action whenever users click a button, and your application should do this whenever the user presses the keyboard equivalent of a button click.

For modal dialog boxes that contain editable text items, your application should handle menu bar access to allow use of your Edit menu and its Cut, Copy, Paste, Clear, and Undo commands. Your event filter function should then test for and handle clicks in your Edit menu and keyboard equivalents for the appropriate commands in your Edit menu. Your application should respond by using the procedures `DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete` to support the Cut, Copy, Paste, and Clear commands.

For an alert box, you specify a pointer to your event filter function in a parameter that you pass to the `Alert`, `StopAlert`, `CautionAlert`, and `NoteAlert` functions. For a modal dialog box, specify a pointer to your event filter function in a parameter that you pass to the `ModalDialog` procedure.

SEE ALSO

Listing 6-27 on page 6-88 illustrates an event filter function. The functions `GetNextEvent` and `SetSystemEventMask` are described in the chapter "Event Manager" in this book.

Resources

This section describes resources used by the Dialog Manager for displaying alerts and dialog boxes. These resources are

- the dialog (`'DLOG'`) resource, which specifies the window type, display rectangle, and item list resource for a dialog box
- the alert (`'ALRT'`) resource, which specifies alert sounds, a display rectangle, and an item list resource for an alert box
- the item list (`'DITL'`) resource, which specifies the items—such as buttons and static text—to display in an alert box or a dialog box
- the dialog color table (`'dctb'`) resource, which lets you supply a color graphics port for a dialog box and also use colors other than the default colors in a dialog box
- the alert color table (`'actb'`) resource, which lets you use colors other than the default colors in an alert box
- the item color table (`'ictb'`) resource, which lets you change the default colors, typeface, font style, and font size of items in an alert box or a dialog box

This section describes the structures of these resources after they are compiled by the Rez resource compiler, available from APDA. If you are interested in creating the Rez input files for these resources, see "Using the Dialog Manager" beginning on page 6-17 for detailed information.

The Dialog Resource

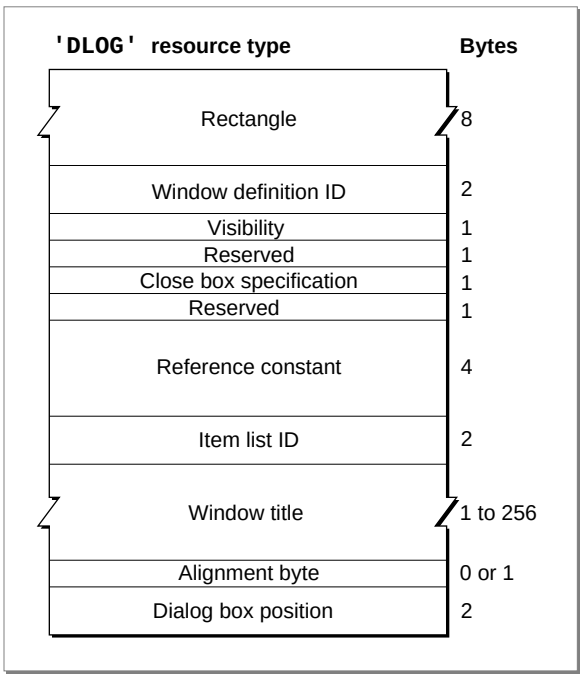
You can use a dialog resource to define a dialog box. A dialog resource is a resource of type `'DLOG'`. All dialog resources must be marked purgeable, and they must have resource ID numbers greater than 128.

To specify the items in a dialog box, you must also provide an item list resource, described beginning on page 6-151. Use the `GetNewDialog` function (described on page 6-113) to create the dialog box defined in the dialog resource.

The format of a Rez input file for a dialog resource differs from its compiled output format. This section describes the structure of a Rez-compiled dialog resource. If you are concerned only with creating a dialog resource, see "Creating Dialog Boxes" beginning on page 6-23.

Figure 6-42 shows the format of a compiled dialog resource.

Figure 6-42 Structure of a compiled dialog (`'DLOG'`) resource



The compiled version of a dialog resource contains the following elements:

- Rectangle. This determines the dialog box's dimensions and, possibly, its position. (The last element in the dialog resource usually specifies a position for the dialog box.)
- Window definition ID.
  - If the integer 0 appears here (as specified in the Rez input file by the `dBoxProc` window definition ID), the Dialog Manager displays a modal dialog box.

- If the integer 4 appears here (as specified in the Rez input file by the `noGrowDocProc` window definition ID), the Dialog Manager displays a modeless dialog box.
- If the integer 5 appears here (as specified in the Rez input file by the `movableDBoxProc` window definition ID), the Dialog Manager displays a movable modal dialog box.

These types of dialog boxes are illustrated in Figure 6-6 on page 6-10, Figure 6-8 on page 6-12, and Figure 6-7 on page 6-11, respectively.

- Visibility. If this is set to a value of 1 (as specified by the `visible` constant in the Rez input file), the Dialog Manager displays this dialog box as soon as you call the `GetNewDialog` function. If this is set to a value of 0 (as specified by the `invisible` constant in the Rez input file), the Dialog Manager does not display this dialog box until you call the Window Manager procedure `ShowWindow`.
- Close box specification. This specifies whether to draw a close box. Normally, this is set to a value of 1 (as specified by the `goAway` constant in the Rez input file) only for a modeless dialog box to specify a close box in its title bar. Otherwise, this is set to a value of 0 (as specified by the `noGoAway` constant in the Rez input file).
- Reference constant. This contains any value that an application stores here. For example, an application can store a number that represents a dialog box type, or it can store a handle to a record that maintains state information about the dialog box or other window types. An application can use the Window Manager procedure `SetWRefCon` at any time to change this value in the dialog record for a dialog box, and you can use the `GetWRefCon` function to determine its current value.
- Item list resource ID. The ID of the item list resource that specifies the items—such as buttons and static text—to display in the dialog box.
- Window title. This is a Pascal string displayed in the dialog box's title bar only when the dialog box is modeless.
- Alignment byte. This is an extra byte added if necessary to make the previous Pascal string end on a word boundary.
- Dialog box position. This specifies the position of the dialog box on the screen. (If your application positions dialog boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager.)
  - If 0x0000 appears here (as specified by the `noAutoCenter` constant in the Rez input file), the Dialog Manager positions this dialog box according to the global coordinates specified in the rectangle element of this resource.
  - If 0xB00A appears here (as specified by the `alertPositionParentWindow` constant in the Rez input file), the Dialog Manager positions the dialog box over the frontmost window so that the window's title bar appears. This is illustrated in Figure 6-33 on page 6-63.
  - If 0x300A appears here (as specified by the `alertPositionMainScreen` constant in the Rez input file), the Dialog Manager centers the dialog box near the top of the main screen. This is illustrated in Figure 6-34 on page 6-63.
  - If 0x700A appears here (as specified in the Rez input file by the `alertPositionParentWindowScreen` constant), the Dialog Manager positions the dialog box on the screen where the user is currently working. This is illustrated in Figure 6-35 on page 6-64.

The Alert Resource

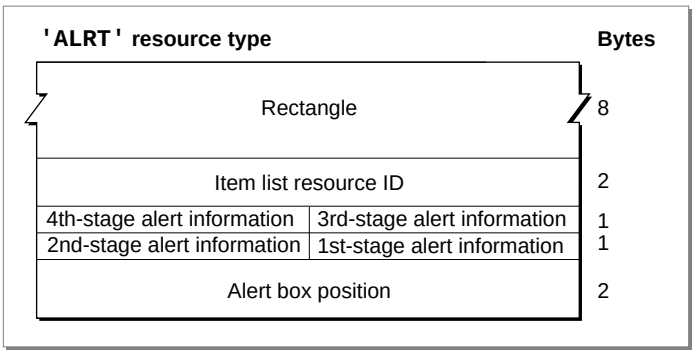
You can use an alert resource to define an alert. An alert resource is a resource of type `'ALRT'`. All alert resources must be marked purgeable, and they must have resource ID numbers greater than 128.

To specify the items in an alert box, you must also provide an item list resource, described beginning on page 6-151. To display the alert, you call either the `NoteAlert`, `CautionAlert`, `StopAlert`, or `Alert` function and pass it the resource ID of the alert resource. The `NoteAlert`, `CautionAlert`, `StopAlert`, and `Alert` functions are described in "Creating Alerts" beginning on page 6-105.

The format of a Rez input file for an alert resource differs from its compiled output format. This section describes the structure of a Rez-compiled alert resource. If you are concerned only with creating an alert resource, see "Creating Alert Sounds and Alert Boxes" beginning on page 6-18.

Figure 6-43 shows the structure of a compiled alert resource.

Figure 6-43 Structure of a compiled alert (`'ALRT'`) resource



The compiled version of an alert resource contains the following elements:

- Rectangle. This determines the alert box's dimensions and, possibly, its position. (The last element in the alert resource usually specifies a position for the alert box.)
- Item list resource ID. The ID of the item list resource that specifies the items—such as buttons and static text—to display in the alert box.
- Fourth-stage alert information. This specifies the response when the user repeats the action that invokes this alert four or more consecutive times. The Dialog Manager responds in the manner specified in the 4 bits that make up this element.
  - If the first bit is set, the Dialog Manager draws a bold outline around the second item in the item list resource (typically, the Cancel button) and—if your application does not specify an event filter function—returns 2 when the user presses the Return or Enter key at the fourth consecutive occurrence of the alert. If the first bit is not set, the Dialog Manager draws a bold outline around the first item in the item list resource (typically, the OK button) and—if your application does not specify an event filter function—returns 1 when the user presses the Return or Enter key.

- If the second bit is set, the Dialog Manager displays the alert box at this stage. If the second bit is not set, the Dialog Manager doesn't display the alert box at this stage.
- If neither of the next 2 bits is set, the Dialog Manager plays no alert sound at this stage. If bit 3 is set and bit 4 is not set, the Dialog Manager plays the first alert sound—by default, the system alert sound. If bit 3 is not set and bit 4 is set, the Dialog Manager plays the second alert sound; by default, it plays the system alert sound twice. If both bit 3 and bit 4 are set, the Dialog Manager plays the third alert sound; by default, it plays the system alert sound three times. By defining your own alert sound (described on page 6-144) and calling the `ErrorSound` procedure (described on page 6-104) to make it the current sound procedure, you can specify your own alert sounds.

- Third-stage alert information. This specifies the response when the user repeats the action that invokes this alert three consecutive times. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- Second-stage alert information. This specifies the response when the user repeats the action that invokes this alert two consecutive times. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- First-stage alert information. This specifies the response for the first time that the user performs the action that invokes this alert. The Dialog Manager interprets these 4 bits in the manner described for the fourth-stage alert.
- Alert box position. This specifies the position of the alert box on the screen. (If your application positions alert boxes on its own, don't use these constants, because your code may conflict with the Dialog Manager.)
  - If 0x0000 appears here (as specified by the `noAutoCenter` constant in the Rez input file), the Dialog Manager positions this alert box according to the global coordinates specified in the rectangle element of this resource.
  - If 0xB00A appears here (as specified by the `alertPositionParentWindow` constant in the Rez input file), the Dialog Manager positions the alert box over the frontmost window so that the window's title bar appears. This is illustrated in Figure 6-33 on page 6-63.
  - If 0x300A appears here (as specified by the `alertPositionMainScreen` constant in the Rez input file), the Dialog Manager centers the alert box near the top of the main screen. This is illustrated in Figure 6-34 on page 6-63.
  - If 0x700A appears here (as specified in the Rez input file by the `alertPositionParentWindowScreen` constant), the Dialog Manager positions the alert box on the screen where the user is currently working. This is illustrated in Figure 6-35 on page 6-64.

The Item List Resource

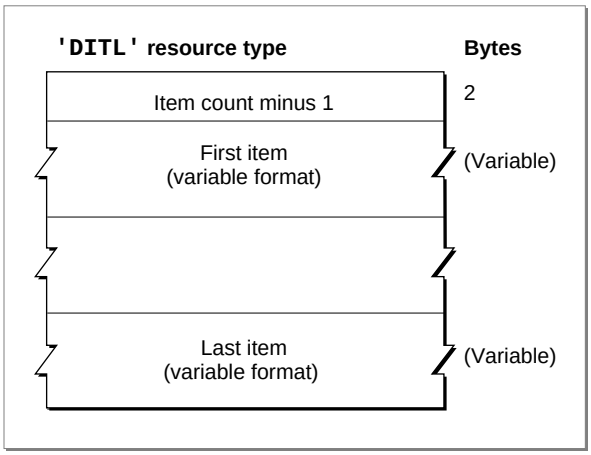
You use an item list resource to specify items—such as buttons and text—in alert boxes and dialog boxes. An item list resource is a resource with the resource type `'DITL'`. All item list resources must be marked purgeable, and they must have resource ID numbers greater than 128.

For an alert box, you specify the resource ID of the item list resource in an alert resource (described beginning on page 6-150). For a dialog box that you create with the `GetNewDialog` function, you specify the resource ID of the item list resource in a dialog resource (described beginning on page 6-148). For a dialog box that you create with either the `NewColorDialog` function (described on page 6-115) or the `NewDialog` function (described on page 6-118), you use the Resource Manager function `GetResource` to read the item list resource into memory and to provide a handle to the item list resource in memory.

The format of a Rez input file for an item list resource differs from its compiled output format. This section describes the structure of a Rez-compiled item list resource. If you are concerned only with creating an item list resource, see "Providing Items for Alert and Dialog Boxes" beginning on page 6-26.

Figure 6-44 shows the format of a compiled item list resource.

Figure 6-44 Structure of a compiled item list (`'DITL'`) resource



The compiled version of an item list resource contains the following elements:

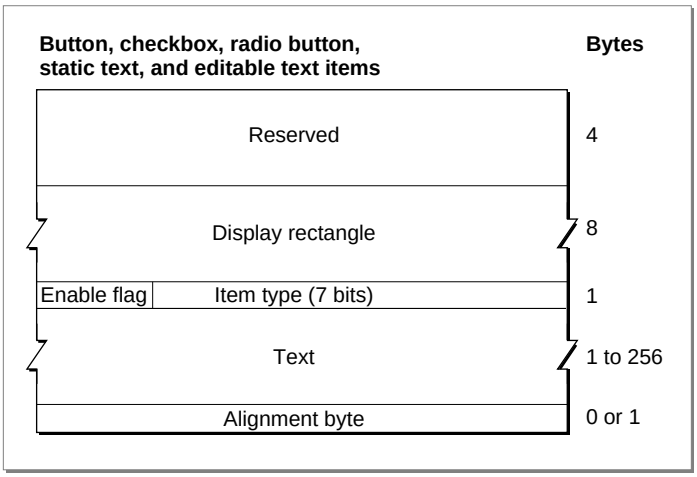
- Item count minus 1. This value is 1 less than the total number of items defined in this resource.
- A variable number of items.

The format of each item depends on its type. Figure 6-45 shows the format of an item defined to be a button, a checkbox, a radio button, a static text item, or an editable text item.

The compiled version of a button, checkbox, radio button, static text item, or editable text item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert box or dialog box; these coordinates specify the upper-left and lower-right corners of the item.

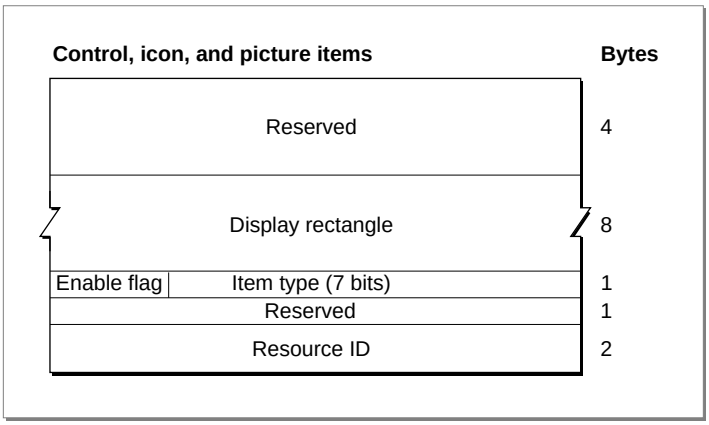
Figure 6-45 Structure of compiled button, checkbox, radio button, static text, and editable text items



- Enable flag. This specifies whether the item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type.
  - If this bit string is set to 4 (as specified in the Rez input file by the `Button` constant), then the item is a button.
  - If this bit string is set to 5 (as specified in the Rez input file by the `CheckBox` constant), then the item is a checkbox.
  - If this bit string is set to 6 (as specified in the Rez input file by the `RadioButton` constant), then the item is a radio button.
  - If this bit string is set to 8 (as specified in the Rez input file by the `StaticText` constant), then the item is static text.
  - If this bit string is set to 16 (as specified in the Rez input file by the `EditText` constant), then the item is editable text.
- Text. This specifies the text that appears in the item. This element consists of a length byte and as many as 255 additional bytes for the text. ("Titles for Buttons, Checkboxes, and Radio Buttons" beginning on page 6-37 and "Text Strings for Static Text and Editable Text Items" beginning on page 6-40 contain recommendations about appropriate text in items.)
  - For a button, checkbox, or radio button, this is the title for that control.
  - For a static text item, this is the text of the item.
  - For an editable text item, this can be an empty string (in which case the editable text item contains no text), or it can be a string that appears as the default string in the editable text item.
- Alignment byte. This is added if necessary to make the previous text string end on a word boundary.

Figure 6-46 shows the format for an element defined to be a control, an icon, or a picture item.

Figure 6-46 Structure of compiled control, icon, and picture items

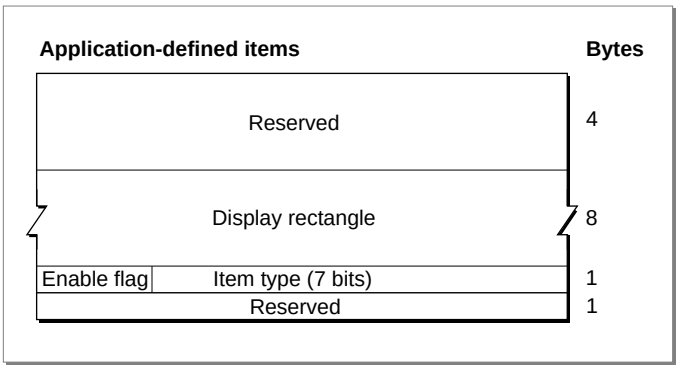


The compiled version of a control, an icon, or a picture item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert or dialog box.
- Enable flag. This specifies whether the item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type.
  - If this 7-bit string is set to 7 (as specified in the Rez input file by the `Control` constant), then the item is a button.
  - If this is set to 32 (as specified in the Rez input file by the `Icon` constant), then the item is an icon.
  - If this is set to 64 (as specified in the Rez input file by the `Picture` constant), then the item is a QuickDraw picture.
- Resource ID.
  - For a control item, this is the resource ID of a `'CTRL'` resource.
  - For an icon item, this is the resource ID of an `'ICON'` resource and, optionally, a `'cicn'` resource
  - For a picture item, this is the resource ID of a `'PICT'` resource.

Figure 6-47 shows the format for an application-defined item.

Figure 6-47 Structure of a compiled application-defined item

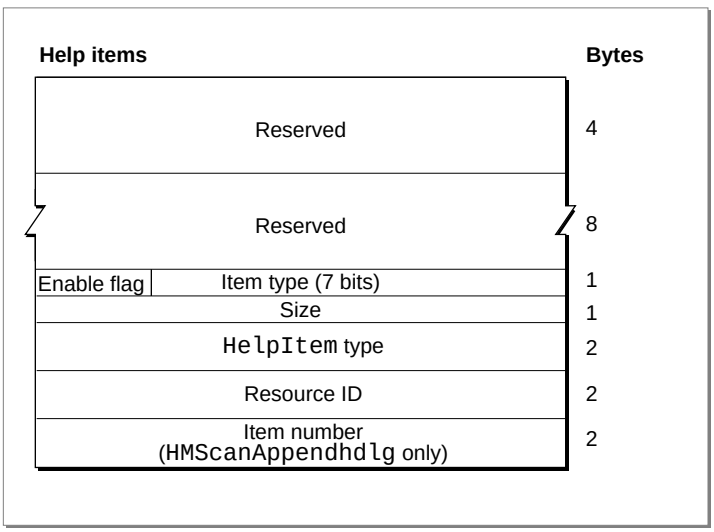


The compiled version of an application-defined item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Display rectangle. This determines the size and location of the application-defined item in the alert box or dialog box. The display rectangle is specified in coordinates local to the alert box or dialog box.
- Enable flag. This specifies whether the application-defined item is enabled or disabled. If this bit is set, the item is enabled and the Dialog Manager reports to your application whenever mouse-down events occur inside this item.
- Item type. This is set to a value of 0 (as specified in the Rez input file by the `UserItem` constant).

Figure 6-48 shows the format for a help item. (Help items are described in detail in the chapter "Help Manager" of *Inside Macintosh: More Macintosh Toolbox*.)

Figure 6-48 Structure of compiled help items



The compiled version of a help item consists of the following elements:

- Reserved. The Dialog Manager uses the element for storage.
- Reserved. This should be set to 0.
- Enable flag. This specifies whether the item is enabled or disabled. For help items, this bit should never be set, because the Dialog Manager cannot report to your application when mouse-down events occur inside the item.
- Item type. This is set to 1 (as specified in the Rez input file by the `HelpItem` constant).
- Size. This specifies the number of bytes contained in the rest of this element. This is set to 4 for an item identified by either the `HMScanhdlg` or `HMScanhrct` identifier, or it's set to 6 for an item identified by the `HMScanAppendhdlg` identifier.
- `HelpItem` type. This specifies the type of help item defined in the resource.
  - For an item identified by the `HMScanhdlg` identifier, this element contains the value 1.
  - For an item identified by the `HMScanhrct` identifier, this element contains the value 2.
  - For an item identified by the `HMScanAppendhdlg` identifier, this element contains the value 8.
- Resource ID. This is the resource ID of the resource containing the help messages for this alert box or dialog box.
  - For an item identified by either the `HMScanhdlg` or `HMScanAppendhdlg` identifier, this is the ID of an `'hdlg'` resource.
  - For an item identified by the `HMScanhrct` identifier, this is the ID of an `'hrct'` resource.
- Item number. This is available only for an item identified by the `HMScanAppendhdlg` identifier. This is the item number within the alert box or dialog box after which the help messages specified in the `'hdlg'` resource should be displayed. These help messages relate to the items that are appended to the alert box or dialog box. (The item list resource does not contain these 2 bytes for items identified by either the `HMScanhdlg` or `HMScanhrct` identifier.)

The Dialog Color Table Resource

On color monitors, the Dialog Manager automatically adds color to your alert and dialog boxes so that they match the colors of the windows, alert boxes, and dialog boxes used by system software. These colors provide aesthetic consistency across all monitors, from black-and-white displays to 8-bit color displays. On a color monitor, for example, the racing stripes in the title bar of a modeless dialog box are gray, the close box and window frame are in color, and the buttons and text are black.

When you create dialog resources, your application's dialog boxes use the system's default colors. Typically, this is all you need to do to provide color for your dialog boxes—with the following exceptions:

- When you need to include a color version of an icon in a dialog box, you must create a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource. Plate 2 at the front of this book shows an alert box that includes a color icon.

- When you need to produce a blended gray color for outlining the inactive (that is, dimmed) default button, you must create a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource.

"Using an Application-Defined Item to Draw the Bold Outline for a Default Button" beginning on page 6-56 explains how to create a draw routine that outlines the default button of a dialog box. If you deactivate a dialog box, you should dim its buttons and use gray to draw the outline for the default button. Because `GetNewDialog` and `NewDialog` supply black-and-white graphics ports for dialog boxes, you can create a dialog color table resource for the dialog box to force the Dialog Manager to supply a color graphics port. Then you can use a blended gray color for the outline for the default button. (The `NewColorDialog` function supplies a color graphics port.)

Even when you create a dialog color table resource for drawing a gray outline, you should not change the system's default colors. If you feel absolutely compelled to use nonstandard colors, you can use the Dialog Manager to specify colors other than the default colors. Your application can specify its own colors for a dialog box by creating a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource (described beginning on page 6-148). You don't have to call any new routines to change the colors used in dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load a dialog color table resource with the same resource ID as the dialog resource.

Be aware, however, that nonstandard colors in your dialog boxes may initially confuse your users. Also be aware that despite any changes you may make, users can alter the colors of dialog boxes anyway by changing settings in the Color control panel.

▲ **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. ▲

A dialog color table resource has exactly the same format as a window color table (that is, a resource of type `'wctb'`), which is described in the chapter "Window Manager" of this book.

If the dialog box's content color isn't white, specify the `invisible` constant in the dialog resource. Use the Window Manager procedure `ShowWindow` to display the dialog box when it's the frontmost window. If the dialog box is a modeless dialog box that is not in front, use the Window Manager procedure `ShowHide` to display it.

The Alert Color Table Resource

On color monitors, the Dialog Manager automatically adds color to your alert boxes so that they match the colors of the windows and alerts used by system software. When you create alert resources, your application's alert boxes use the system's default colors. Typically, this is all you need to do to provide color for your alert boxes. (However, to include a color version of an icon in an alert box, you must add a resource of type `'cicn'` with the same resource ID as the black-and-white `'ICON'` resource specified in the item list resource.)

If you feel absolutely compelled to use nonstandard colors, you can use the Dialog Manager to specify colors other than the default colors. Your application can specify its own colors for an alert box by creating an alert color table (`'actb'`) resource with the same resource ID as the alert resource (described beginning on page 6-150). You don't have to call any new routines to change the colors used in alert or dialog boxes. When you call the `Alert` function, for example, the Dialog Manager automatically attempts to load an alert color table resource with the same resource ID as the alert resource.

Be aware, however, that nonstandard colors in your alert boxes may initially confuse your users. Also be aware that despite any changes you may make, users can alter the colors of dialog boxes anyway by changing settings in the Color control panel.

▲ **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. ▲

An alert color table resource has exactly the same format as a window color table (`'wctb'`) resource, which is described in the chapter "Window Manager" of this book.

The Item Color Table Resource

On color monitors, the Dialog Manager automatically draws the items in your dialog and alert boxes so that they match the colors of the items used by system software in its dialog and alert boxes. The Dialog Manager also uses the default system font when it draws the text in the static text and editable text items of your dialog and alert boxes.

If you feel absolutely compelled to use nonstandard fonts and colors, you can use the Dialog Manager to specify your own colors, typeface, font style, and font size.

Note
The Dialog Manager displays the typeface, font style, and font size you specify only on color monitors. ◆

Your application can specify these by creating an item color table (`'ictb'`) resource with the same resource ID as the dialog or alert box's item list resource , and then providing a dialog color table resource for a dialog box or an alert color table resource for an alert box. You don't have to call any new routines to change the colors, typefaces, font styles, or font sizes used in dialog boxes. When you call the `GetNewDialog` function, for example, the Dialog Manager automatically attempts to load an item color table resource with the same resource ID as the item list resource.

Note
To make it easier to localize your application for other script systems, you should not change the font. Do not use a smaller font, such as 9-point Geneva; some script systems, such as KanjiTalk, require 12-point fonts. ◆

Also, be aware that nonstandard colors for items in your dialog and alert boxes may initially confuse your users.

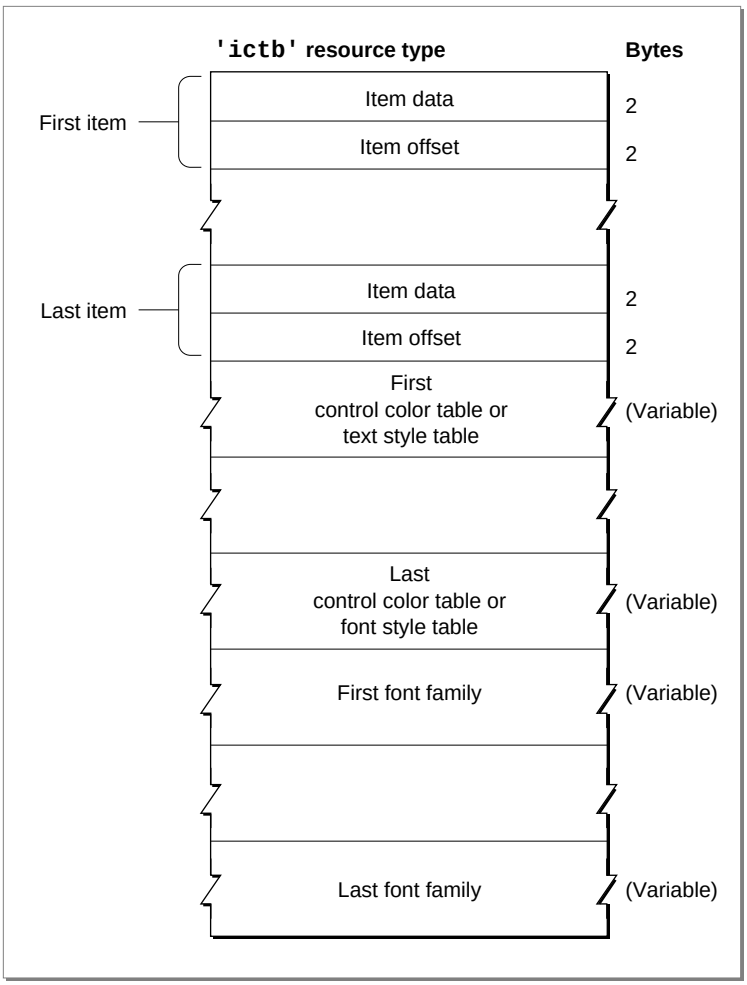
▲ **WARNING**
Because the behavior of color alert and dialog boxes, color items, and color icons is unreliable on computers using system software versions earlier than System 7, do not create these color elements if you wish to maintain backward compatibility. ▲

If you want to provide an item color table resource for an alert box or a dialog box, you must create an alert color table resource or a dialog color table resource, even if the item color table resource has no actual color information and describes only static text and editable text style changes.

An item color table resource is a resource of type `'ictb'`. All item color table resources must have resource ID numbers greater than 128.

There is no Rez template available for creating item color table resources. When you compile an item color table resource, it should follow the format illustrated in Figure 6-49.

Figure 6-49 Structure of a compiled item color table resource



You define an item color table resource for a dialog box or an alert box by specifying these elements in a resource with the `'ictb'` resource type:

- Items. These consist of a variable number of items, corresponding to those in an item list resource with the same resource ID as this item color table resource.
- Control color tables and text style tables.
  - A **control color table** defines the colors used in a control. Several controls can share the same control color table.
  - A **text style table** defines the font family, font style, font size, and color of text in an editable text item or a static text item. Several editable text and static text items can share the same text style table.
- Optionally, a list of font families. If you use any text style tables, you generally conclude the item color table resource with a list of text strings, each of which specifies a font family. Although you may specify font numbers instead of font names, it's much more reliable to specify names, because system software may renumber these fonts as they are installed and removed. For every editable text item and static text item listed at the top of the item color table resource, specify a font family at the bottom of the resource.

The information contained in an element depends on the type of item it describes:

- Item data. This contains information about how this item is described in the rest of this resource.
  - For a control, this is the length (in bytes) of its control color table.
  - For a static text item or an editable text item, the bits of this element determine which elements of the text style table to use and are interpreted as follows:

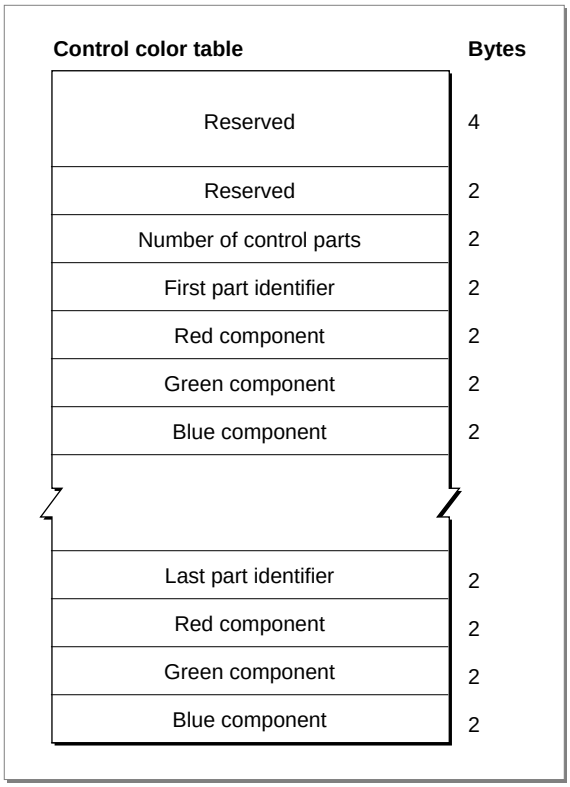
Bit	Meaning
0	Change the font family.
1	Change the typeface.
2	Change the font size.
3	Change the font foreground color.
4	Add the font size.
13	Change the font background color.
14	Change the font mode.
15	The font element is an offset to the name.

- Item offset. The number of bytes from the beginning of the resource to either the control color table or the text style table that describes this item.

When both the item data and item offset elements are set to 0, then the control or text item is drawn with the default colors, typeface, font size, and font style. Even if only the first few items of the dialog box have color style information, there must be room for all of the items actually in the box (with the item data and item offset elements of the unused entries set to 0).

For controls, the colors are described by a color table identical to a `'cctb'` resource used by the Control Manager. Multiple controls can use the same color table. If the resource sets both the item data and the item offset element to 0, then the system's default colors are used for the control. The format of a control color table is illustrated in Figure 6-50.

Figure 6-50 Structure of a compiled control color table



A control color table consists of the following elements:

- Reserved. This should always be set to a value of 0.
- Reserved. Again, should always be set to a value of 0.
- Number of control parts. For standard controls other than scroll bars, this should be set to 3, because a standard control uses only three parts: frame, control body, and text. For scroll bars, this should be set to 12; see the description of the control color table resource in the chapter "Control Manager" for information on specifying the colors for a scroll bar. To create a control that uses other parts, you must create a custom `'CDEF'` resource, as described in the chapter "Control Manager" in this book.

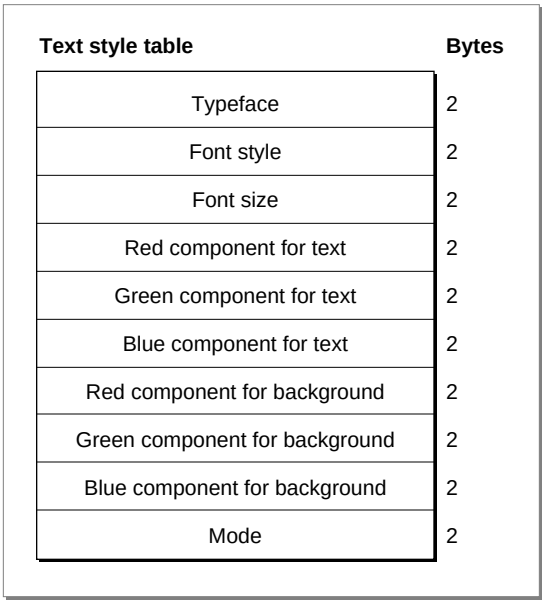
- Part identifier. This is a value that identifies a part of the first control. The following list shows the values and constants they represent for the standard controls other than scroll bars. For information on the part identifiers for a scroll bar, see the description of the control color table resource in the chapter "The Control Manager" in this book. They can be listed in any order in the control color table.

Constant	Value	Control part
`cFrameColor`	0	Frame
`cBodyColor`	1	Body
`cTextColor`	2	Text (such as titles)

- Red component. This is an integer that represents the intensity of the red component of the color to use when drawing this control part.
- Green component. This is an integer that represents the intensity of the green component of the color to use when drawing this control part.
- Blue component. This is an integer that represents the intensity of the blue component of the color to use when drawing this control part.
- Part identifier, and the red, green, and blue color components for the next control part. Specify color components for every part of this control whose color you want to change. If a part is not listed in the control color table, the Dialog Manager draws it in its default color.

Figure 6-51 shows the format of a text style table.

Figure 6-51 Structure of a compiled text style table



The text style table must be 20 bytes long, as shown in Figure 6-51. Multiple editable text and static text items can use the same text style record. To display text in the standard typeface, color, font size, and font style, set the item data and item offset elements for the item to 0. Allocate space for all fields in the text style table, even if they are not used.

A text style table consists of the following elements (see *Inside Macintosh: Text* for a discussion of font families, font style, and point sizes):

- Typeface. This is the name of the font family to use. If bit 15 in the item data element is set to 1, then this element contains an offset (in bytes) to a font name element at the end of the resource. If bit 0 in the item data element is set to 1, then this element contains the number of a font family. If bit 0 in the item data element is set to 0, this element is set to 0, and the system default font is used.
- Font style. This is the font style to use. If bit 1 in the item data element is set to 1, then this element uses the bits of the low-order byte to describe which styles to apply to the text. If all bits in the low-order byte are set to 0, the plain font style is used. The bit numbers and the styles they represent are

Bit value	Style
0	Bold
1	Italic
2	Underline
3	Outline
4	Shadow
5	Condensed
6	Extended

- Font size. This is the point size of the font. If bit 2 in the item data element is set to 1, this element contains a value representing a point size. If bit 4 in the item data element is set to 1, this element contains a value to add to the current point size of the text. If bit 0 in the item data element is set to 0, this element is set to 0, and the system font size (12) is used.
- Text red color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the red component of the color to use when drawing the text.
- Text green color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the green component of the color to use when drawing the text.
- Text blue color. If bit 3 in the item data element is set to 1, this element contains an integer that represents the intensity of the blue component of the color to use when drawing the text.
- Background red color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the red component of the color to use when drawing the background behind the text.

- Background green color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the green component of the color to use when drawing the background behind the text.
- Background blue color. If bit 13 in the item data element is set to 1, this element contains an integer that represents the intensity of the blue component of the color to use when drawing the background behind the text.
- Mode. If bit 14 in the item data element is set to 1, this element contains an integer that represents how characters are placed in the bit image. The values that the Dialog Manager interprets and the constants that represent them are listed here. See *Inside Macintosh: Imaging* for a discussion of source transfer modes.

Constant	Value
`scrOr`	1
`srcXor`	2
`srcBic`	3

Summary of the Dialog Manager

Pascal Summary

Constants

```
CONST
{checking for AppendDITL, ShortenDITL, CountDITL using Gestalt function}
   gestaltDITLExtAttr = 'ditl';  {Gestalt selector for AppendDITL, etc.}
   gestaltDITLExtPresent = 0;    {if this bit's set, then AppendDITL, }
                                 { ShortenDITL, & CountDITL are available}

{item types for GetDialogItem, SetDialogItem}
   ctrlItem       = 4;     {add this constant to the next four constants}
   btnCtrl        = 0;     {standard button control}
   chkCtrl        = 1;     {standard checkbox control}
   radCtrl        = 2;     {standard radio button}
   resCtrl        = 3;     {control defined in a control resource}
   helpItem       = 1;     {help balloons}
   statText       = 8;     {static text}
   editText       = 16;    {editable text}
   iconItem       = 32;    {icon}
   picItem        = 64;    {QuickDraw picture}
   userItem       = 0;     {application-defined item}
   itemDisable    = 128;   {add to any of the above to disable it}

{item numbers of OK and Cancel buttons in alert boxes}
   ok       = 1;  {first button is OK button}
   cancel   = 2;  {second button is Cancel button}

{resource IDs of alert box icons}
   stopIcon    = 0;
   noteIcon    = 1;
   cautionIcon = 2;

{constants used for theMethod parameter in AppendDITL}
   overlayDITL       = 0;  {overlay existing items}
   appendDITLRight   = 1;  {append at right}
   appendDITLBottom  = 2;  {append at bottom}
```

```
{constants for procID parameter of NewDialog, NewColorDialog}
   dBoxProc          = 1;  {modal dialog box}
   noGrowDocProc     = 4;  {modeless dialog box}
   movableDBoxProc   = 5;  {movable modal dialog box}
```

Data Types

```
TYPE  DialogPtr      =  WindowPtr;
ResumeProcPtr        =  ProcPtr;
SoundProcPtr         =  ProcPtr;
ModalFilterProcPtr   =  ProcPtr;
DialogPeek           =  ^DialogRecord;
DialogRecord         =
RECORD
   window:     WindowRecord;  {dialog window}
   items:      Handle;        {item list resource}
   textH:      TEHandle;      {current editable text item}
   editField:  Integer;       {editable text item number minus 1}
   editOpen:   Integer;       {used internally}
   aDefItem:   Integer;       {default button item number}
END;

DITLMethod = Integer;
```

Dialog Manager Routines

Initializing the Dialog Manager

```
PROCEDURE InitDialogs       (resumeProc: ResumeProcPtr);
PROCEDURE ErrorSound        (soundProc: SoundProcPtr);
PROCEDURE SetDialogFont     (fontNum: Integer); {also spelled SetDAFont}
```

Creating Alerts

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
FUNCTION Alert              (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION StopAlert          (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION NoteAlert          (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION CautionAlert       (alertID: Integer; filterProc:
                             ModalFilterProcPtr): Integer;
FUNCTION GetAlertStage      : Integer;
PROCEDURE ResetAlertStage;
```

Creating and Disposing of Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
FUNCTION GetNewDialog       (dialogID: Integer; dStorage: Ptr;
                             behind:  WindowPtr): DialogPtr;
FUNCTION NewColorDialog     (dStorage: Ptr; boundsRect: Rect; title:
                             Str255; visible: Boolean; procID: Integer;
                             behind: WindowPtr; goAwayFlag: Boolean;
                             refCon: LongInt; items: Handle): DialogPtr;
FUNCTION NewDialog          (dStorage:  Ptr; boundsRect: Rect; title:
                             Str255; visible: Boolean; procID: Integer;
                             behind: WindowPtr; goAwayFlag: Boolean;
                             refCon: LongInt; items: Handle):  DialogPtr;
PROCEDURE CloseDialog       (theDialog: DialogPtr);
PROCEDURE DisposeDialog     (theDialog: DialogPtr);
```

Manipulating Items in Alert and Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE GetDialogItem     (theDialog: DialogPtr; itemNo: Integer;
                             VAR itemType: Integer; VAR item: Handle;
                             VAR box: Rect);
PROCEDURE SetDialogItem     (theDialog: DialogPtr; itemNo: Integer;
                             itemType: Integer; item: Handle; box: Rect);
PROCEDURE HideDialogItem    (theDialog: DialogPtr; itemNo: Integer);
PROCEDURE ShowDialogItem    (theDialog: DialogPtr; itemNo: Integer);
FUNCTION FindDialogItem     (theDialog: DialogPtr; thePt: Point): Integer;
PROCEDURE AppendDITL        (theDialog: DialogPtr; theDITL: Handle;
                             theMethod: DITLMethod);
PROCEDURE ShortenDITL       (theDialog: DialogPtr; numberItems: Integer);
FUNCTION CountDITL          (theDialog: DialogPtr): Integer;
```

Handling Text in Alert and Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE ParamText         (param0: Str255; param1: Str255;
                             param2: Str255; param3: Str255);
PROCEDURE GetDialogItemText (item: Handle; VAR text: Str255);
PROCEDURE SetDialogItemText (item: Handle; text: Str255);
PROCEDURE SelectDialogItemText
                            (theDialog: DialogPtr; itemNo: Integer;
                             strtSel: Integer; endSel: Integer);
PROCEDURE DialogCut         (theDialog: DialogPtr);
PROCEDURE DialogCopy        (theDialog: DialogPtr);
```

```
PROCEDURE DialogPaste       (theDialog: DialogPtr);
PROCEDURE DialogDelete      (theDialog: DialogPtr);
```

Handling Events in Dialog Boxes

```
{some routines have 2 spellings--see Table 6-1 for the alternate spellings}
PROCEDURE ModalDialog       (filterProc: ModalFilterProcPtr; VAR itemHit:
                             Integer);
FUNCTION IsDialogEvent      (theEvent: EventRecord): Boolean;
FUNCTION DialogSelect       (theEvent: EventRecord; VAR theDialog:
                             DialogPtr; VAR itemHit: Integer): Boolean;
PROCEDURE DrawDialog        (theDialog: DialogPtr);
PROCEDURE UpdateDialog      (theDialog: DialogPtr; updateRgn: RgnHandle);
```

Application-Defined Routines

```
PROCEDURE MyItem            (theWindow: WindowPtr; itemNo: Integer);
PROCEDURE MyAlertSound      (soundNo: Integer);
FUNCTION MyEventFilter      (theDialog: DialogPtr; VAR theEvent:
                             EventRecord; VAR itemHit: Integer): Boolean;
```

C Summary

Constants

```
enum {
/*checking for AppendDITL, ShortenDITL, CountDITL using Gestalt function*/
   #define gestaltDITLExtAttr 'ditl' /*Gestalt selector*/
   gestaltDITLExtPresent = 0     /*if this bit's set, then AppendDITL, */
                                 /* ShortenDITL, & CountDITL are available*/
};

enum {
/*item types for GetDItem, SetDItem*/
   ctrlItem       = 4,  /*add this constant to the next four constants*/
   btnCtrl        = 0,  /*standard button control*/
   chkCtrl        = 1,  /*standard checkbox control*/
   radCtrl        = 2,  /*standard radio button*/
   resCtrl        = 3,  /*control defined in a control resource*/
   statText       = 8,  /*static text*/
   editText       = 16, /*editable text*/
   iconItem       = 32, /*icon*/
```

```
   picItem        = 64, /*QuickDraw picture*/
   userItem       = 0,  /*application-defined item*/
   helpItem       = 1,  /*help balloons*/
   itemDisable    = 128,/*add to any of the above to disable it*/

/*item numbers of OK and Cancel buttons in alert boxes*/
   ok       = 1,  /*first button is OK button*/
   cancel   = 2,  /*second button is Cancel button*/

/*resource IDs of alert box icons*/
   stopIcon    = 0,
   noteIcon    = 1,
   cautionIcon = 2
};

enum {
/*constants used for theMethod parameter in AppendDITL*/
   overlayDITL       = 0,  /*overlay existing items*/
   appendDITLRight   = 1,  /*append at right*/
   appendDITLBottom  = 2   /*append at bottom*/
};

enum {
/*constants for procID parameter of NewDialog, NewColorDialog*/
   dBoxProc          = 1,  /*modal dialog box*/
   noGrowDocProc     = 4,  /*modeless dialog box*/
   movableDBoxProc   = 5   /*movable modal dialog box*/
};
```

Data Types

```
typedef WindowPtr DialogPtr;

typedef struct DialogRecord DialogRecord;
typedef struct DialogRecord *DialogPeek;

struct DialogRecord{
      WindowRecord   window;     /*dialog window*/
      Handle         items;      /*item list resource*/
      TEHandle       textH;      /*current editable text item*/
      short          editField;  /*editable text item number minus 1*/
      short          editOpen;   /*used internally*/
      short          aDefItem;   /*default button item number*/
};
```

```
typedef pascal void (*ResumeProcPtr)(void);
typedef pascal void (*SoundProcPtr)(void);
typedef pascal Boolean (*ModalFilterProcPtr)(DialogPtr theDialog,
                                    EventRecord *theEvent, short *itemHit);
typedef short DITLMethod;
```

Dialog Manager Routines

Initializing the Dialog Manager

```
pascal void InitDialogs     (ResumeProcPtr resumeProc);
pascal void ErrorSound      (SoundProcPtr soundProc);
pascal void SetDialogFont   (short fontNum); /*also spelled SetDAFont*/
```

Creating Alerts

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal short Alert          (short alertID, ModalFilterProcPtr filterProc);
pascal short StopAlert      (short alertID, ModalFilterProcPtr filterProc);
pascal short NoteAlert      (short alertID, ModalFilterProcPtr filterProc);
pascal short CautionAlert   (short alertID, ModalFilterProcPtr filterProc);
#define GetAlertStage()     (* (short*) 0x0A9A);
pascal void ResetAlertStage (void);
```

Creating and Disposing of Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal DialogPtr GetNewDialog
                            (short dialogID, void *dStorage,
                             WindowPtr behind);
pascal DialogPtr NewColorDialog
                            (void *dStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short procID, WindowPtr behind,
                             Boolean goAwayFlag, long refCon, Handle items);
pascal DialogPtr NewDialog
                            (void *dStorage, const Rect *boundsRect,
                             ConstStr255Param title, Boolean visible,
                             short procID, WindowPtr behind,
                             Boolean goAwayFlag, long refCon,
                             Handle items);
pascal void CloseDialog     (DialogPtr theDialog);
pascal void DisposeDialog   (DialogPtr theDialog);
```

Manipulating Items in Alert and Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void GetDialogItem   (DialogPtr theDialog, short itemNo,
                             short *itemType, Handle *item, Rect *box);
pascal void SetDialogItem   (DialogPtr theDialog, short itemNo, short
                             itemType, Handle item, const Rect *box);
pascal void HideDialogItem  (DialogPtr theDialog, short itemNo);
pascal void ShowDialogItem  (DialogPtr theDialog, short itemNo);
pascal short FindDialogItem (DialogPtr theDialog, Point thePt);
pascal void AppendDITL      (DialogPtr theDialog, Handle theDITL,
                             DITLMethod theMethod);
pascal void ShortenDITL     (DialogPtr theDialog, short numberItems);
pascal short CountDITL      (DialogPtr theDialog);
```

Handling Text in Alert and Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void ParamText       (ConstStr255Param param0,
                             ConstStr255Param param1,
                             ConstStr255Param param2,
                             ConstStr255Param param3);
pascal void GetDialogItemText
                            (Handle item, Str255 text);
pascal void SetDialogItemText
                            (Handle item, ConstStr255Param text);
pascal void SelectDialogItemText
                            (DialogPtr theDialog, short itemNo,
                             short strtSel, short endSel);
pascal void DialogCut       (DialogPtr theDialog);
pascal void DialogCopy      (DialogPtr theDialog);
pascal void DialogPaste     (DialogPtr theDialog);
pascal void DialogDelete    (DialogPtr theDialog);
```

Handling Events in Dialog Boxes

```
/*some routines have 2 spellings--see Table 6-1 for the alternate spellings*/
pascal void ModalDialog     (ModalFilterProcPtr filterProc, short *itemHit);
pascal Boolean IsDialogEvent(const EventRecord *theEvent);
pascal Boolean DialogSelect (const EventRecord *theEvent,
                             DialogPtr *theDialog, short *itemHit);
pascal void DrawDialog      (DialogPtr theDialog);
pascal void UpdateDialog    (DialogPtr theDialog, RgnHandle updateRgn);
```

Application-Defined Routines

```
pascal void MyItem          (WindowPtr theWindow, short itemNo);
pascal void MyAlertSound    (short soundNo);
pascal Boolean MyEventFilter(DialogPtr theDialog, *EventRecord theEvent,
                             *short itemHit);
```

Assembly-Language Summary

Data Structures

DialogRecord Data Structure

0	`dWindow`	156 bytes	window record for the alert box or dialog box
156	`items`	long	handle to the item list resource for the alert box or dialog box
160	`teHandle`	long	handle to the current editable text item
164	`editField`	word	current editable text item
166	`editOpen`	word	used internally
168	`aDefItem`	word	item number of the default button

Global Variables

`DAStrings`	Handles to text strings specified with the `ParamText` procedure
`DABeeper`	Address of current sound procedure
`DlgFont`	Font number for text in dialog boxes and alert boxes
`ACount`	Alert stage number (0 through 3) of the last alert
`ANumber`	Resource ID of last alert
`ResumeProc`	Address of resume procedure (should not be used in System 7)