INSIDE MACINTOSH

Mac OS 8 Toolbox Reference

Updated for Appearance 1.0.1

December 10, 1997
Technical Publications

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to the Mac OS 8 Toolbox Reference

Mac OS 8 Toolbox Reference describes the elements of the Mac OS Toolbox that are new as of Mac OS 8 or Appearance 1.0.1, as well as those earlier parts of the Toolbox that have been affected by Mac OS 8 or Appearance 1.0.1.

In cases where Mac OS 8's features, particularly the Appearance Manager and contextual menus, have altered the use of preexisting elements of the Toolbox, a discussion of their prior implementation and functionality (going back to System 7) is also included. This document does not provide pre–System 7 Toolbox support.

IMPORTANT

This document only includes types, constants, resources, and functions that are new as of Mac OS 8, changed (in implementation or form) as of Mac OS 8, or not recommended as of Mac OS 8. Unaffected information has not been included in this document; therefore, if a function, for example, is not presented here, its use can be assumed to be unchanged.

The following categories of reference material are included in this document:

- For those Toolbox elements that have changed with Mac OS 8, you will find the new information, as well as that portion of the System 7 description which is still valid as of Mac OS 8. This will be followed by the information that is no longer applicable as of Mac OS 8, but which may still be useful for those developers who must maintain System 7 code or develop for systems lacking the Appearance Manager.
- For Toolbox elements that are no longer recommended as of Mac OS 8, you will see the conditions under which the item is superseded as well as the recommended alternatives to the item's use.
- For new Toolbox elements, you will simply be provided with their implementation and use as of Mac OS 8.

About This Document

This document contains the following seven chapters and an index:

- Chapter 1, "Appearance Manager Reference," describes the Appearance Manager and its functions, resources, types, and constants.
- Chapter 2, "Control Manager Reference," presents the usage of the new Mac OS 8 controls as well as describes how older controls have changed with the Appearance Manager.
- Chapter 3, "Window Manager Reference," discusses how the creation and manipulation of windows has changed as of Mac OS 8 due to the Appearance Manager.
- Chapter 4, "Dialog Manager Reference," explains how Mac OS 8 and the Appearance Manager have changed the handling of dialog and alert boxes.
- Chapter 5, "Menu Manager Reference," shows how your application's handling of menus and menu bars has been modified by Mac OS 8, the Appearance Manager, and contextual menus.
- Chapter 6, "Event Manager Reference," is unchanged by Mac OS 8 from its earlier form in *Macintosh Toolbox Essentials*, and it is therefore not presented in this delta document.
- Chapter 7, "Finder Interface Reference," describes how Mac OS 8's new Folder Manager features have affected how your application works with the Finder.

Format of a Typical Chapter

A typical chapter will follow this structure:

- "Types and Constants." This section describes the types, constants, structures, and result codes for the functions described later in the chapter.
- "Resources." The resource descriptions contain complete listings of their elements, provided in Rez format or as a figure.
- "Functions." This section presents function declarations, descriptions of parameters and function results, and discussions of function use.

Development Environment

All constants, type definitions, and function declarations in this book are written in C or C++ and are Power PC–compliant.

The system software functions described in this book are available using C, Pascal, or assembly-language interfaces. How you access these functions depends on the development environment you are using. When showing system software functions, this book uses the C interface available with Universal Interfaces 3.0.

For More Information

For a full description of the System 7 Toolbox elements that are unchanged as of Mac OS 8, as well as assembly-language and pre–System 7 Mac OS Toolbox information, see *Inside Macintosh: Macintosh Toolbox Essentials* and *More Macintosh Toolbox*, available at your local bookseller.

For information on the Mac OS 8 human interface, see *Mac OS 8 Human Interface Guidelines*, available at the Apple Developer World web site. Apple Developer World is the best source for finding the most up-to-date technical and marketing information specifically for developers of Macintosh-compatible software and hardware products. Developer World can be found at:

<http://www.devworld.apple.com>

Conventions Used

This document uses the following conventions to alert you to especially important categories of information.

Quick Reference Tags

The following tags appear underneath section headings in this document. They are provided to alert you to how a given function, type, resource, or constant has been affected by Mac OS 8, Appearance, or contextual menus.

NEW WITH . . .

Functions, resources, types, and constants that have been introduced with Mac OS 8, Appearance, or contextual menus.

IMPORTANT

Many of the Toolbox functions discussed are dependent upon the presence of the Appearance Manager. Unless otherwise specified, it is assumed that any function described as "New With the Appearance Manager" depends upon the availability of the Appearance Manager and that you should check for the presence of the Appearance Manager before calling that function. See the chapter "Appearance Manager Reference" for details on using Gestalt selector constants to determine whether the Appearance Manager is present and its version, if so.

CHANGED WITH . . .

Functions, resources, types, and constants that have been changed, but are still used. The changes will be discussed in the main body of the text, and the functionality on systems lacking Mac OS 8, Appearance, or contextual menus will be summarized at the end of the description.

NOT RECOMMENDED WITH . . .

Functions, resources, types, and constants that are not recommended to be used with Mac OS 8, Appearance, or contextual menus. Alternate recommendations are presented in lieu of the description.

Special Fonts

All code listings, reserved words, and names of actual data structures, fields, constants, parameters, and functions are shown in Letter Gothic `(this is Letter Gothic)`.

Empty Strings

This book occasionally instructs you to provide an empty string in function parameters and resources. How you specify an empty string depends on what language and development environment you are using. In Rez input files and in C, for example, you specify an empty string by two double quotation marks ("").

Types of Notes

There are several types of notes used in this document.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text.

IMPORTANT
A note like this contains information that is essential for an understanding of the main text.

▲ **WARNING**
Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data.

Appearance Manager Reference

Contents

This chapter discusses the Appearance Manager, versions 1.0 and 1.0.1.

- "Appearance Manager Types and Constants" (page 21) describes the Appearance Manager types and constants. Result codes are included at the end of this section.
- "Appearance Manager Functions" (page 34) describes Appearance Manager functions that you can call to initialize the Appearance Manager, draw Appearance-compliant menus and other human interface elements, and coordinate the colors and patterns of human interface elements.

Introduction to the Appearance Manager

The Appearance Manager

- coordinates the look of the Mac OS human interface into a single theme
- introduces new human interface elements to the Mac OS environment
- allows for the adaptation of pre–Appearance Manager human interface elements, both standard and custom, to the new, coordinated look and behaviors

The Appearance Manager provides the underlying support for themes and theme switching. **Themes** unify the appearance and behavior of human interface objects in your application, including alert icons, controls, background colors, dialog boxes, menus, windows, and state transitions. The only theme supported under Appearance 1.0 and 1.0.1 is the platinum appearance.

IMPORTANT

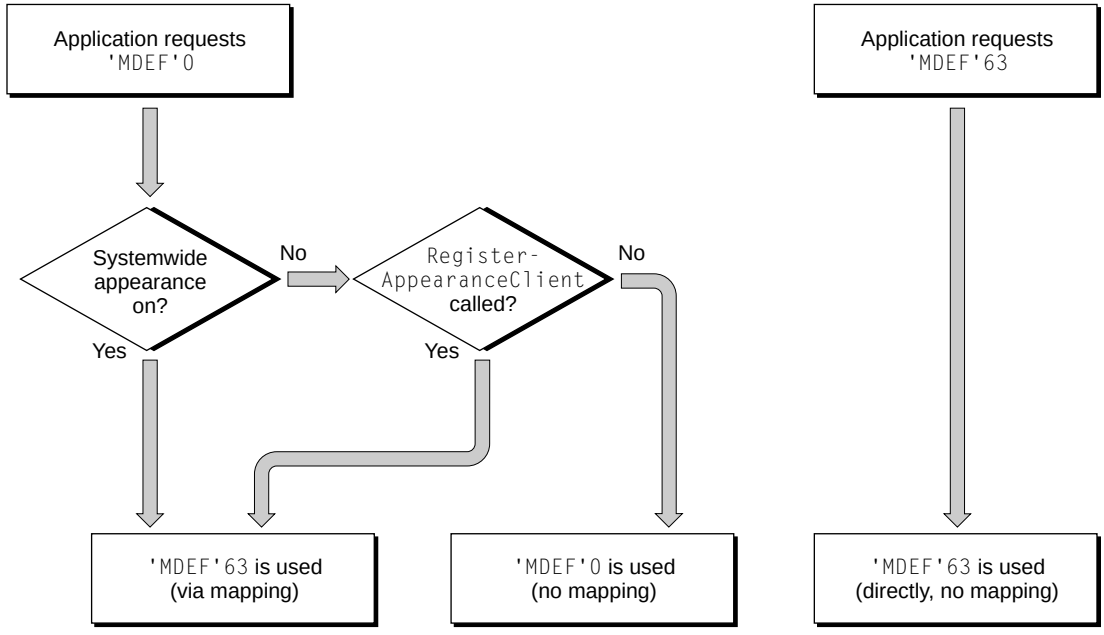
Appearance 1.0 ships with and supports Mac OS 8 only. It should not normally be used with earlier Mac OS versions. Appearance 1.0.1 is an extension that is designed to work with versions of the Mac OS platform from System 7.1 on, so it supersedes Appearance 1.0.

To provide a systemwide coordination of look and behavior, the Appearance Manager includes new standard human interface objects that were developed to replace the many custom solutions that have been implemented. These new

elements, such as focus rings and group boxes, obviate the need for developers to create and maintain their own.

Another way the Appearance Manager achieves a unified look and behavior is by **mapping** standard pre-Appearance definition functions (the `'MBDF'0`, `'MDEF'0`, `'WDEF'0`, `'WDEF'124`, `'CDEF'0`, `'CDEF'1`, and `'CDEF'63` resources) to their Appearance-compliant equivalents. This occurs either on a systemwide basis (if the user hasn't turned off systemwide appearance) or on a per-application basis, if you call `RegisterAppearanceClient` (page 34) from within your application. Figure 1-1 shows the ways by which it is determined how, and whether, mapping will occur for standard definition functions.

Figure 1-1 Mapping of standard definition functions



Some mapped definition functions will have a slightly different look and behavior than if they were specified directly. For example, since a standard pre-Appearance window definition function can't specify the inclusion of a horizontal zoom box, when the old resource is mapped to a new one, the

resulting window still won't have a horizontal zoom box. For this reason (and to eliminate the time spent going through the mapping layer), it's recommended that you specify the Appearance-compliant definition function IDs directly.

Custom definition functions cannot be automatically mapped to Appearance-compliant equivalents. However, the Appearance Manager does provide ways to coordinate custom elements with themes. For example, using `DrawThemeListBoxFrame` (page 42) creates a theme-compliant frame for a custom list box.

Appearance Manager Types and Constants

Appearance Manager Gestalt Selector Constants

NEW WITH THE APPEARANCE MANAGER

Before calling any functions dependent upon the Appearance Manager's presence, your application should pass the selector `gestaltAppearanceAttr` to the `Gestalt` function to determine whether the Appearance Manager is present.

```
enum {
    gestaltAppearanceAttr           = 'appr'
};
```

Constant description

`gestaltAppearanceAttr`
The `Gestalt` selector passed to determine whether the Appearance Manager is present. Produces a 32-bit value whose bits you should test to determine which Appearance Manager features are available.

The following values are the bit numbers with which you can test for the presence of each feature:

```
enum {
    gestaltAppearanceExists             = 0,
    gestaltAppearanceCompatMode         = 1
};
```

Constant descriptions

`gestaltAppearanceExists`
: If this bit (bit 0) is set, Appearance Manager functions are available. To determine which version of the Appearance Manager is installed, check for the presence of the `Gestalt` selector `gestaltAppearanceVersion` (see below). If this bit is not set, Appearance Manager functions are not available.

`gestaltAppearanceCompatMode`
: If this bit (bit 1) is set, systemwide platinum appearance is off, turning off auto-mapping of standard System 7 definition functions to their Mac OS 8 equivalents (for those applications that have not called `RegisterAppearanceClient`). If this bit is not set, systemwide platinum appearance is on, causing auto-mapping of standard System 7 definition functions to their Mac OS 8 equivalents for all applications.

To determine which version of the Appearance Manager is installed, your application should check for the presence of the `Gestalt` selector `gestaltAppearanceVersion`.

```
enum{
    gestaltAppearanceVersion        = 'apvr'
};
```

Constant description

`gestaltAppearanceVersion`
: The `Gestalt` selector passed to determine which version of the Appearance Manager is installed. If this selector exists, Appearance Manager 1.0.1 (or later) is installed; the version number of the currently installed Appearance Manager is returned in the low-order word of the result in BCD format (for example, version 1.0.1 would be 0x0101). If this selector does not exist but `gestaltAppearanceAttr` does, Appearance Manager 1.0 is installed.

Appearance Manager Apple Event Constants

NEW WITH THE APPEARANCE MANAGER

The following Apple event constants have been defined under Appearance 1.0.1, but apply to Appearance versions 1.1 and later. They are provided here for informational purposes only.

```
enum {
    kAppearanceEventClass   = 'appr',
    kAEThemeSwitch          =' thme'
};
```

Constant descriptions

`kAppearanceEventClass`
: The event class of Appearance Manager Apple events.

`kAEThemeSwitch`
: The Apple event constant `kAEThemeSwitch` allows you to prepare for Appearance Manager 1.1. Under Appearance 1.1, when the theme is changed in the Appearance control panel, `kAEThemeSwitch` will be sent to all running foreground applications that are high-level event aware. Applications can handle the switch as appropriate.

Appearance-Compliant Brush Type Constants

NEW WITH THE APPEARANCE MANAGER

The Appearance Manager provides the underlying support for `RGB` color data and overrides System 7 color tables such as `'cctb'` and `'mctb'` with a more abstract mechanism that allows colors and patterns to be coordinated with the current theme. You can pass constants of type `ThemeBrush` in the `inBrush` parameter of `SetThemeBackground` (page 56), `SetThemePen` (page 57), and `SetThemeWindowBackground` (page 59) to specify Appearance-compliant colors or patterns for standard human interface elements.

```
enum {
    kThemeActiveDialogBackgroundBrush               = 1,
    kThemeInactiveDialogBackgroundBrush             = 2,
    kThemeActiveAlertBackgroundBrush                = 3,
    kThemeInactiveAlertBackgroundBrush              = 4,
```

```
    kThemeActiveModelessDialogBackgroundBrush       = 5,
    kThemeInactiveModelessDialogBackgroundBrush     = 6,
    kThemeActiveUtilityWindowBackgroundBrush        = 7,
    kThemeInactiveUtilityWindowBackgroundBrush      = 8,
    kThemeListViewSortColumnBackgroundBrush         = 9,
    kThemeListViewBackgroundBrush                   = 10,
    kThemeIconLabelBackgroundBrush                  = 11,
    kThemeListViewSeparatorBrush                    = 12,
    kThemeChasingArrowsBrush                        = 13,
    kThemeDragHiliteBrush                           = 14,
    kThemeDocumentWindowBackgroundBrush             = 15,
    kThemeFinderWindowBackgroundBrush               = 16
};
typedef SInt16 ThemeBrush;
```

Constant descriptions

`kThemeActiveDialogBackgroundBrush`
: An active dialog box's background color or pattern.

`kThemeInactiveDialogBackgroundBrush`
: An inactive dialog box's background color or pattern.

`kThemeActiveAlertBackgroundBrush`
: An active alert box's background color or pattern.

`kThemeInactiveAlertBackgroundBrush`
: An inactive alert box's background color or pattern.

`kThemeActiveModelessDialogBackgroundBrush`
: An active modeless dialog box's background color or pattern.

`kThemeInactiveModelessDialogBackgroundBrush`
: An inactive modeless dialog box's background color or pattern.

`kThemeActiveUtilityWindowBackgroundBrush`
: An active utility window's background color or pattern.

`kThemeInactiveUtilityWindowBackgroundBrush`
: An inactive utility window's background color or pattern.

`kThemeListViewSortColumnBackgroundBrush`
: The background color or pattern of the column upon which a list view is sorted.

`kThemeListViewBackgroundBrush`
: The background color or pattern of a list view column that is not being sorted upon.

`kThemeIconLabelBackgroundBrush`
: An icon label's color or pattern.

`kThemeListViewSeparatorBrush`
: A list view separator's color or pattern.

`kThemeChasingArrowsBrush`
: Asynchronous arrows' color or pattern.

`kThemeDragHiliteBrush`
: The background color or pattern of an element responding to a drag and drop, indicating that the element is a valid recipient.

`kThemeDocumentWindowBackgroundBrush`
: A document window's background color or pattern.

`kThemeFinderWindowBackgroundBrush`
: A Finder window's background color or pattern. Generally, you should not use this constant unless you are trying to create a window that matches the Finder window.

Appearance-Compliant Text Color Constants

NEW WITH THE APPEARANCE MANAGER

You can pass constants of type `ThemeTextColor` in the `inColor` parameter of `SetThemeTextColor` (page 58) to specify Appearance-compliant text colors for many standard human interface elements in their active, inactive, and highlighted states.

```
enum{
    kThemeActiveDialogTextColor                 = 1,
    kThemeInactiveDialogTextColor               = 2,
    kThemeActiveAlertTextColor                  = 3,
    kThemeInactiveAlertTextColor                = 4,
    kThemeActiveModelessDialogTextColor         = 5,
    kThemeInactiveModelessDialogTextColor       = 6,
    kThemeActiveWindowHeaderTextColor           = 7,
    kThemeInactiveWindowHeaderTextColor         = 8,
    kThemeActivePlacardTextColor                = 9,
```

```
    kThemeInactivePlacardTextColor                = 10,
    kThemePressedPlacardTextColor                 = 11,
    kThemeActivePushButtonTextColor               = 12,
    kThemeInactivePushButtonTextColor             = 13,
    kThemePressedPushButtonTextColor              = 14,
    kThemeActiveBevelButtonTextColor              = 15,
    kThemeInactiveBevelButtonTextColor            = 16,
    kThemePressedBevelButtonTextColor             = 17,
    kThemeActivePopupButtonTextColor              = 18,
    kThemeInactivePopupButtonTextColor            = 19,
    kThemePressedPopupButtonTextColor             = 20,
    kThemeIconLabelTextColor                      = 21,
    kThemeListViewTextColor                       = 22,
    kThemeActiveDocumentWindowTitleTextColor      = 23,
    kThemeInactiveDocumentWindowTitleTextColor    = 24,
    kThemeActiveMovableModalWindowTitleTextColor  = 25,
    kThemeInactiveMovableModalWindowTitleTextColor = 26,
    kThemeActiveUtilityWindowTitleTextColor       = 27,
    kThemeInactiveUtilityWindowTitleTextColor     = 28,
    kThemeActivePopupWindowTitleColor             = 29,
    kThemeInactivePopupWindowTitleColor           = 30,
    kThemeActiveRootMenuTextColor                 = 31,
    kThemeSelectedRootMenuTextColor               = 32,
    kThemeDisabledRootMenuTextColor               = 33,
    kThemeActiveMenuItemTextColor                 = 34,
    kThemeSelectedMenuItemTextColor               = 35,
    kThemeDisabledMenuItemTextColor               = 36,
    kThemeActivePopupLabelTextColor               = 37,
    kThemeInactivePopupLabelTextColor             = 38
};
typedef SInt16 ThemeTextColor;
```

Constant descriptions

`kThemeActiveDialogTextColor`
: Text color for active dialog box.

`kThemeInactiveDialogTextColor`
: Text color for inactive dialog box.

`kThemeActiveAlertTextColor`
: Text color for active alert box. The text color for alert boxes may differ from dialog box text color.

`kThemeInactiveAlertTextColor`
: Text color for inactive alert box. The text color for alert boxes may differ from dialog box text color.

`kThemeActiveModelessDialogTextColor`
: Text color for active modeless dialog box.

`kThemeInactiveModelessDialogTextColor`
: Text color for inactive modeless dialog box.

`kThemeActiveWindowHeaderTextColor`
: Text color for active window header.

`kThemeInactiveWindowHeaderTextColor`
: Text color for inactive window header.

`kThemeActivePlacardTextColor`
: Text color for active placard.

`kThemeInactivePlacardTextColor`
: Text color for inactive placard.

`kThemePressedPlacardTextColor`
: Text color for highlighted placard.

`kThemeActivePushButtonTextColor`
: Text color for active push button.

`kThemeInactivePushButtonTextColor`
: Text color for inactive push button.

`kThemePressedPushButtonTextColor`
: Text color for highlighted push button.

`kThemeActiveBevelButtonTextColor`
: Text color for active bevel button.

`kThemeInactiveBevelButtonTextColor`
: Text color for inactive bevel button.

`kThemePressedBevelButtonTextColor`
: Text color for highlighted bevel button.

`kThemeActivePopupButtonTextColor`
: Text color for active pop-up menu button.

`kThemeInactivePopupButtonTextColor`
: Text color for inactive pop-up menu button.

`kThemePressedPopupButtonTextColor`
: Text color for highlighted pop-up menu button.

`kThemeIconLabelTextColor`
: Text color for icon label.

`kThemeListViewTextColor`
Text color for list view.

`kThemeActiveDocumentWindowTitleTextColor`
Text color for active document window title. Available with Appearance 1.0.1 and later.

`kThemeInactiveDocumentWindowTitleTextColor`
Text color for inactive document window title. Available with Appearance 1.0.1 and later.

`kThemeActiveMovableModalWindowTitleTextColor`
Text color for active movable modal window title. Available with Appearance 1.0.1 and later.

`kThemeInactiveMovableModalWindowTitleTextColor`
Text color for inactive movable modal window title. Available with Appearance 1.0.1 and later.

`kThemeActiveUtilityWindowTitleTextColor`
Text color for active utility (floating) window title. Available with Appearance 1.0.1 and later.

`kThemeInactiveUtilityWindowTitleTextColor`
Text color for inactive utility (floating) window title. Available with Appearance 1.0.1 and later.

`kThemeActivePopupWindowTitleColor`
Text color for active pop-up window title. Available with Appearance 1.0.1 and later.

`kThemeInactivePopupWindowTitleColor`
Text color for inactive pop-up window title. Available with Appearance 1.0.1 and later.

`kThemeActiveRootMenuTextColor`
Text color for active root menu. Available with Appearance 1.0.1 and later.

`kThemeSelectedRootMenuTextColor`
Text color for selected root menu. Available with Appearance 1.0.1 and later.

`kThemeDisabledRootMenuTextColor`
Text color for disabled root menu. Available with Appearance 1.0.1 and later.

`kThemeActiveMenuItemTextColor`
Text color for active menu item. Available with Appearance 1.0.1 and later.

`kThemeSelectedMenuItemTextColor`
Text color for inactive menu item. Available with Appearance 1.0.1 and later.

`kThemeDisabledMenuItemTextColor`
Text color for disabled menu item. Available with Appearance 1.0.1 and later.

`kThemeActivePopupLabelTextColor`
Text color for active pop-up menu label. Available with Appearance 1.0.1 and later.

`kThemeInactivePopupLabelTextColor`
Text color for inactive pop-up menu label. Available with Appearance 1.0.1 and later.

Appearance-Compliant Draw State Constants

NEW WITH THE APPEARANCE MANAGER

You can pass constants of type `ThemeDrawState` in the `inState` parameter of functions used for drawing human interface elements to specify whether they are drawn as active (normal), selected (pressed), or inactive (disabled). For descriptions of the functions that use these constants, see "Drawing Appearance-Compliant Controls" (page 35).

```
enum {
    kThemeStateDisabled     = 0,
    kThemeStateActive       = 1,
    kThemeStatePressed      = 2
};
typedef UInt32 ThemeDrawState;
```

Constant descriptions

`kThemeStateDisabled`
Element is drawn in its disabled, inactive state.

`kThemeStateActive`
Element is drawn in its normal, active state.

`kThemeStatePressed`
Element is drawn in its selected, pressed state.

Appearance-Compliant Menu Bar Draw State Constants

NEW WITH THE APPEARANCE MANAGER

You can pass one of the following constants in the `inState` parameter of `DrawThemeMenuBarBackground` (page 45) to specify whether Appearance-compliant menu bars are drawn as normal or selected.

```
enum{
    kThemeMenuBarNormal         = 0,
    kThemeMenuBarSelected       = 1
};
typedef SInt16 ThemeMenuBarState;
```

Constant descriptions

`kThemeMenuBarNormal`
Menu bar is drawn in its normal state.

`kThemeMenuBarSelected`
Menu bar is drawn in its selected state.

If you wish the menu bar to be drawn with square upper corners (as for a laptop system) instead of rounded ones (as for a desktop system), your application should set the bit for the attribute `kThemeMenuSquareMenuBar`.

```
enum {
    kThemeMenuSquareMenuBar= (1 << 0)
};
```

Constant descriptions

`kThemeMenuSquareMenuBar`
Menu bar is drawn with square corners.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before using the `ThemeMenuBarState` type or the Appearance-compliant menu bar draw state constants. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Appearance-Compliant Menu Draw State Constants

NEW WITH THE APPEARANCE MANAGER

You can pass one of the following constants in the `inState` parameter of `DrawThemeMenuItem` (page 48) and `DrawThemeMenuTitle` (page 46) to specify the state in which Appearance-compliant menus are drawn.

```
enum{
    kThemeMenuActive            = 0,
    kThemeMenuSelected          = 1,
    kThemeMenuDisabled          = 2
};
typedef SInt16 ThemeMenuState;
```

Constant descriptions

`kThemeMenuActive` Menu is drawn in its active state.

`kThemeMenuSelected` Menu is drawn in its selected state.

`kThemeMenuDisabled` Menu is drawn in its disabled state.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before using the `ThemeMenuState` type or the Appearance-compliant menu draw state constants. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Appearance-Compliant Menu Type Constants

NEW WITH THE APPEARANCE MANAGER

You can pass one of the following constants in the `inMenuType` parameter of `GetThemeMenuBackgroundRegion` (page 51) and `DrawThemeMenuBackground` (page 48) to specify menu type.

```
enum {
    kThemeMenuTypePullDown      = 0,
    kThemeMenuTypePopUp         = 1,
```

```
    kThemeMenuTypeHierarchical  = 2
};
typedef SInt16 ThemeMenuType;
```

Constant descriptions

`kThemeMenuTypePullDown`
A pull-down menu.

`kThemeMenuTypePopUp`
A pop-up menu.

`kThemeMenuTypeHierarchical`
A hierarchical menu.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before using the `ThemeMenuType` type or the Appearance-compliant menu type constants. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Appearance-Compliant Menu Item Type Constants

NEW WITH THE APPEARANCE MANAGER

You can pass one of the following constants in the `inItemType` parameters of `DrawThemeMenuItem` (page 48) and `GetThemeMenuItemExtra` (page 54) to specify menu item type.

```
enum {
    kThemeMenuItemPlain             = 0,
    kThemeMenuItemHierarchical      = 1,
    kThemeMenuItemScrollUpArrow     = 2,
    kThemeMenuItemScrollDownArrow   = 3
};
typedef SInt16 ThemeMenuItemType;
```

Constant descriptions

`kThemeMenuItemPlain`
A plain menu item.

`kThemeMenuItemHierarchical`
A hierarchical menu item.

`kThemeMenuItemScrollUpArrow`
A scroll-up arrow.

`kThemeMenuItemScrollDownArrow`
A scroll-down arrow.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before using the `ThemeMenuItemType` type or the Appearance-compliant menu item type constants. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Result Codes

The most common result codes that can be returned by Appearance Manager functions are listed below.

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough memory
`appearanceBadBrushIndexErr`	–30560	Invalid brush color constant
`appearanceProcessRegisteredErr`	–30561	Application already registered as Appearance Manager client
`appearanceProcessNotRegisteredErr`	–30562	Application not registered as Appearance Manager client
`appearanceBadTextColorIndexErr`	–30563	Invalid text color constant
`appearanceThemeHasNoAccents`	–30564	Theme does not support accent colors

Appearance Manager Functions

Initializing the Appearance Manager

RegisterAppearanceClient

NEW WITH THE APPEARANCE MANAGER

Makes your application Appearance-compliant and enables the mapping of standard pre-Appearance definition functions to their Appearance-compliant equivalents.

```
pascal OSStatus RegisterAppearanceClient (void);
```

function result A result code. The result code `appearanceProcessRegisteredErr` indicates that your process was already registered when you called the `RegisterAppearanceClient` function. For a list of other result codes, see "Result Codes" (page 33).

DISCUSSION

The `RegisterAppearanceClient` function must be called at the beginning of your application, prior to initializing or drawing any onscreen elements or invoking any definition functions, such as the menu bar. Under Appearance 1.0 and 1.0.1, applications that call `RegisterAppearanceClient` will continue to have a platinum look when systemwide appearance is off.

`RegisterAppearanceClient` automatically maps standard pre-Appearance definition functions to their Appearance-compliant equivalents, whether or not the user has turned on systemwide appearance; see "Introduction to the Appearance Manager" (page 19) for more details on this process. Although they will not make use of mapping, applications that specify Appearance-compliant definition function IDs directly should also call `RegisterAppearanceClient`.

UnregisterAppearanceClient

NEW WITH THE APPEARANCE MANAGER

Makes your application non-Appearance-compliant and turns off the mapping of standard pre-Appearance definition functions to their Appearance-compliant equivalents.

```
pascal OSStatus UnregisterAppearanceClient (void);
```

function result A result code. The result code `appearanceProcessNotRegisteredErr` indicates that your process was not registered when you called the `UnregisterAppearanceClient` function. For a list of other result codes, see "Result Codes" (page 33).

DISCUSSION

The `UnregisterAppearanceClient` function is automatically called for you when your application terminates. Normally this function does not need to be called. You might want to call this function if you are running a plug-in architecture, and you know that a given plug-in isn't Appearance compliant. In this case you would bracket your use of the plug-in with calls to `UnregisterAppearanceClient` (before the plug-in is used) and `RegisterAppearanceClient` (after the plug-in is used), so that Appearance is turned off for the duration of the plug-in's usage.

Drawing Appearance-Compliant Controls

DrawThemePrimaryGroup

NEW WITH THE APPEARANCE MANAGER

Draws a primary group box frame consistent with the current theme.

```
pascal OSStatus DrawThemePrimaryGroup (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect`	On input, a pointer to a rectangle.
`inState`	A value specifying the state in which the primary group box frame is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The frame can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The primary group box frame is drawn inside the rectangle that is passed and is a maximum of 2 pixels thick.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeSecondaryGroup

NEW WITH THE APPEARANCE MANAGER

Draws a secondary group box frame consistent with the current theme.

```
pascal OSStatus DrawThemeSecondaryGroup (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect`	On input, a pointer to a rectangle.
`inState`	A value specifying the state in which the secondary group box frame is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The frame can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeSecondaryGroup` function allow you to nest a secondary group box frame within the primary group box frame. The secondary group box frame is drawn inside the rectangle that is passed and is a maximum of 2 pixels thick.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeSeparator

NEW WITH THE APPEARANCE MANAGER

Draws a separator line consistent with the current theme.

```
pascal OSStatus DrawThemeSeparator (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect`	On input, a pointer to a rectangle.
`inState`	A value specifying the state in which the separator line is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The separator line can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeSeparator` function draws a separator line inside the rectangle passed in, which is a maximum of 2 pixels thick. The orientation of the rectangle determines where the separator line is drawn. If the rectangle is wider than it is tall, the separator line is horizontal; otherwise it is vertical.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeWindowHeader

NEW WITH THE APPEARANCE MANAGER

Draws a window header consistent with the current theme.

```
pascal OSStatus DrawThemeWindowHeader (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect` On input, a pointer to a rectangle.

`inState` A value specifying the state in which the window header is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The header can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeWindowHeader` function draws a window header such as that used by the Finder. The window header is drawn inside the rectangle that is passed.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeWindowListViewHeader

NEW WITH THE APPEARANCE MANAGER

Draws a window list view header consistent with the current theme.

```
pascal OSStatus DrawThemeWindowListViewHeader (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect` On input, a pointer to a rectangle.

`inState`	A value specifying the state in which the window list view header is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The header can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeWindowListViewHeader` function draws a window list view header, such as that used by the Finder, inside the rectangle that is passed in. A window list view header is drawn without a line on its bottom edge, so that bevel buttons can be placed against it without overlapping.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemePlacard

NEW WITH THE APPEARANCE MANAGER

Draws a placard consistent with the current theme.

```
pascal OSStatus DrawThemePlacard (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect`	On input, a pointer to a rectangle.
`inState`	A value specifying the state in which the placard is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The placard can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemePlacard` function draws a placard inside the rectangle that is passed.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeModelessDialogFrame

NEW WITH THE APPEARANCE MANAGER

Draws a modeless dialog box frame consistent with the current theme.

```
pascal OSStatus DrawThemeModelessDialogFrame (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect` On input, a pointer to a rectangle.

`inState` A value specifying the state in which the modeless dialog box frame is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The frame can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeModelessDialogFrame` function draws a modeless dialog box frame, like the one drawn by the Dialog Manager. This function may be used to make a custom modeless dialog box Appearance-compliant.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeModelessDialogFrame` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeEditTextFrame

NEW WITH THE APPEARANCE MANAGER

Draws an editable text frame consistent with the current theme.

```
pascal OSStatus DrawThemeEditTextFrame (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect` On input, a pointer to a rectangle.

`inState` A value specifying the state in which the editable text frame is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The frame can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeEditTextFrame` function draws an editable text frame (a maximum of 2 pixels thick) outside the rectangle passed in. The rectangle should be the same as the one passed in the function `DrawThemeFocusRect` (page 43), so you can get the correct focus look for your editable text control. You should not use these frames for items other than editable text fields.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeListBoxFrame

NEW WITH THE APPEARANCE MANAGER

Draws a list box frame consistent with the current theme.

```
pascal OSStatus DrawThemeListBoxFrame (
                    const Rect *inRect,
                    ThemeDrawState inState);
```

`inRect` On input, a pointer to a rectangle.

`inState` A value specifying the state in which the list box frame is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The frame can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeListBoxFrame` function draws a list box frame (a maximum of 2 pixels thick) outside the rectangle passed in. The rectangle should be the same as the one passed into the function `DrawThemeFocusRect` (page 43) so you can get the correct focus look for your list box control.

SEE ALSO

"Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeGenericWell

NEW WITH THE APPEARANCE MANAGER

Draws an image well frame consistent with the current theme.

```
pascal OSStatus DrawThemeGenericWell (
                    const Rect *inRect,
                    ThemeDrawState inState,
                    Boolean inFillCenter);
```

`inRect`	On input, a pointer to a rectangle.
`inState`	A value specifying the state in which the image well frame is to be drawn; see "Appearance-Compliant Draw State Constants" (page 29). The well can only be drawn as active or inactive; passing `kThemeStatePressed` will result in an error being returned.
`inFillCenter`	A Boolean value indicating whether the image well frame is to be filled in with white.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeGenericWell` function draws an image well frame, for use with custom image well controls. You can specify that the center of the well be filled in with white.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeGenericWell` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

DrawThemeFocusRect

NEW WITH THE APPEARANCE MANAGER

Draws or erases a focus ring around a specified rectangle consistent with the current theme.

```
pascal OSStatus DrawThemeFocusRect (
                    const Rect *inRect,
                    Boolean inHasFocus);
```

`inRect`	On input, a pointer to a rectangle.
`inHasFocus`	A Boolean value. If `true`, the focus ring should be drawn. If `false`, the focus ring should be erased.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeFocusRect` function should be used to indicate that an item has keyboard focus. The focus ring is drawn outside the rectangle that is passed in and can be outset a maximum of 3 pixels.

SPECIAL CONSIDERATIONS

To achieve the right look, you should first call `DrawThemeEditTextFrame` (page 41) or `DrawThemeListBoxFrame` (page 42) and then call `DrawThemeFocusRect`, passing the same rectangle in the `inRect` parameter. If you use `DrawThemeFocusRect` to erase the focus ring around an editable text frame or list box frame, you will have to redraw the editable text frame or list box frame because there is typically an overlap.

SEE ALSO

"Dialog Box Guidelines" in *Mac OS 8 Human Interface Guidelines*.

DrawThemeFocusRegion

NEW WITH THE APPEARANCE MANAGER

Draws or erases an Appearance-compliant focus ring around a specified region.

```
pascal OSStatus DrawThemeFocusRegion (
                    RgnHandle inRegion,
                    Boolean inHasFocus);
```

`inRegion` On input, a handle to a region.

`inHasFocus` A Boolean value. If `true`, the focus region should be drawn. If `false`, the focus region should be erased.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeFocusRegion` function draws or erases a region to receive keyboard focus.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeFocusRegion` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Drawing Appearance-Compliant Menus

DrawThemeMenuBarBackground

NEW WITH THE APPEARANCE MANAGER

Draws an Appearance-compliant menu bar background.

```
pascal OSStatus DrawThemeMenuBarBackground (
                    const Rect *inBounds,
                    ThemeMenuBarState inState,
                    UInt32 inAttributes);
```

`inBounds` On input, a pointer to a rectangle providing global coordinates that specify the menu bar's initial size and location.

`inState` A value specifying the state (active or selected) in which the menu bar is to be drawn; see "Appearance-Compliant Menu Bar Draw State Constants" (page 30).

`inAttributes` Reserved. Pass 0.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

Use the `DrawThemeMenuBarBackground` function if you are writing a custom menu bar definition function and wish to coordinate with the current theme.

An Appearance-compliant menu bar background is drawn in the rectangle passed in the `inBounds` parameter.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeMenuBarBackground` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

DrawThemeMenuTitle

NEW WITH THE APPEARANCE MANAGER

Draws an Appearance-compliant menu title.

```
pascal OSStatus DrawThemeMenuTitle (
                    const Rect *inMenuBarRect,
                    const Rect *inTitleRect,
                    ThemeMenuState inState,
                    UInt32 inAttributes,
                    MenuTitleDrawingUPP inTitleProc,
                    UInt32 inTitleData);
```

`inMenuBarRect` On input, a pointer to a rectangle, which should contain the entire menu bar that the title is being drawn into. The menu bar background is drawn in the rectangle passed in the `inMenuBarRect` parameter. Call `GetThemeMenuBarHeight` (page 52) to get the height of the menu bar.

`inTitleRect` On input, a pointer to a rectangle, which should contain the entire title. The title background is drawn in the rectangle passed in the `inTitleRect` parameter. The width of this rectangle is determined by calculating the width of the menu title's content and then calling `GetThemeMenuTitleExtra` (page 55) to get the amount of padding between menu titles in the current theme; these two values are added together and added to the left edge of where the title should be drawn. The top and bottom coordinates of this rectangle should be the same as those of the `inMenuBarRect` parameter.

`inState`	A value specifying the state (active, selected, or disabled) in which the menu title is to be drawn; see "Appearance-Compliant Menu Draw State Constants" (page 31).
`inAttributes`	Reserved. Pass 0.
`inTitleProc`	On input, a pointer to a menu title drawing function such as `MyMenuTitleDrawingProc` (page 61), defining how to draw the contents of the menu title. The value of the `inTitleProc` parameter can be a valid universal procedure pointer or `nil`.
`inTitleData`	Data to be passed in to the `inUserData` parameter of `MyMenuTitleDrawingProc` (page 61). This data is usually a pointer to information needed to draw the title's contents, such as a pointer to a string.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeMenuTitle` function should be called when you are writing your own custom menu bar definition function and wish to coordinate with the current theme. Your menu title drawing function will be called clipped to the rectangle in which the menu title content is drawn; do not draw outside this region. At the time your menu title drawing function is called, the foreground text color and mode is already set to draw text in the specified state (enabled, selected, or disabled) and correct color for the theme. You do not need to set the color unless you have special drawing needs.

IMPORTANT

You should not depend on the background color for your menu title, so you should not call the `EraseRect` function from your menu title drawing function.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeMenuTitle` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

DrawThemeMenuBackground

NEW WITH THE APPEARANCE MANAGER

Draws an Appearance-compliant menu background.

```
pascal OSStatus DrawThemeMenuBackground (
                    const Rect *inMenuRect,
                    ThemeMenuType inMenuType);
```

`inMenuRect` On input, a pointer to a rectangle, which should contain the entire menu.

`inMenuType` A value specifying the type of menu (pull-down, pop-up, or hierarchical) for which a background is being drawn; see "Appearance-Compliant Menu Type Constants" (page 31).

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeMenuBackground` function draws a menu background in the rectangle specified; it should be called when you are writing a custom menu bar definition function and wish to coordinate with the current theme.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeMenuBackground` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

DrawThemeMenuItem

NEW WITH THE APPEARANCE MANAGER

Draws a menu item that coordinates with the current theme.

```
pascal OSStatus DrawThemeMenuItem (
                    const Rect *inMenuRect,
                    const Rect *inItemRect,
```

```
                    SInt16 inVirtualMenuTop,
                    SInt16 inVirtualMenuBottom,
                    ThemeMenuState inState,
                    ThemeMenuItemType inItemType,
                    MenuItemDrawingUPP inDrawProc,
                    UInt32 inUserData);
```

`inMenuRect` On input, a pointer to a rectangle that should contain the area of the entire menu; this is the actual menu rectangle as used in your menu definition function.

`inItemRect` On input, a pointer to a rectangle, which should contain the area of the entire menu item. The menu item's background will be drawn in the rectangle passed in the `inItemRect` parameter. You should calculate the size of the menu item's content and then call `GetThemeMenuItemExtra` (page 54) to get the amount of padding surrounding menu items in the current theme; the width and height of the menu item rectangle are determined by adding these values together.

`inVirtualMenuTop`
An integer representing the true top of the menu. Normally this value is the top coordinate of the rectangle supplied in the `inMenuRect` parameter. This value could be different, however, if a menu is scrolled or bigger than can be displayed in the menu rectangle. You would normally pass the value of the global variable `TopMenuItem` into this parameter if you are writing a custom menu definition function.

`inVirtualMenuBottom`
An integer representing the true bottom of the menu. Normally this value is the bottom coordinate of the rectangle supplied in the `inMenuRect` parameter. This value could be different, however, if a menu is scrolled or bigger than can be displayed in the menu rectangle. You would normally pass the value of the global variable `AtMenuBottom` into this parameter if you are writing a custom menu definition function.

`inState` A value specifying the state (active, selected, or disabled) in which the menu item is to be drawn; see "Appearance-Compliant Menu Draw State Constants" (page 31).

`inItemType`	A constant of type `ThemeMenuItemType`. If you pass `kThemeMenuItemScrollUpArrow` or `kThemeMenuItemScrollDownArrow`, then pass `nil` for the `inDrawProc` parameter, since there's no content to be drawn. If you pass `kThemeMenuItemHierarchical`, the hierarchical arrow is drawn for you. See "Appearance-Compliant Menu Item Type Constants" (page 32).
`inDrawProc`	On input, a pointer to your menu item drawing function. The value of the `inDrawProc` parameter can be a valid universal procedure pointer or `nil`; see `MyMenuItemDrawingProc` (page 63).
`inUserData`	Data to be passed in to the `inUserData` parameter of `MyMenuItemDrawingProc` (page 63). This data is usually a pointer to information needed to draw the item's contents, such as a pointer to a string.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeMenuItem` function should be called when you are writing your own custom menu definition function and wish to coordinate menu items with the current theme. Your menu drawing function will be called clipped to the rectangle in which you are allowed to draw your content; do not draw outside this region. At the time your menu drawing function is called, the foreground text color and mode are already set to draw text in the specified state (enabled, selected, disabled) and correct color for the theme. You do not need to set the color unless you have special drawing needs.

IMPORTANT

You should not depend on the background color for your menu item, so you should not call the `EraseRect` function from your menu item drawing function.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeMenuItem` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

DrawThemeMenuSeparator

NEW WITH THE APPEARANCE MANAGER

Draws an Appearance-compliant menu item separator line.

```
pascal OSStatus DrawThemeMenuSeparator (const Rect *inItemRect);
```

`inItemRect` On input, a pointer to the rectangle in which the menu item separator should be drawn. The rectangle passed should be the same height as the height returned by the function `GetThemeMenuSeparatorHeight` (page 53).

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `DrawThemeMenuSeparator` function should be called when you are writing your own menu bar definition function and wish to coordinate a menu item separator line with the current theme.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `DrawThemeMenuSeparator` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

GetThemeMenuBackgroundRegion

NEW WITH THE APPEARANCE MANAGER

Gets the background region a menu occupies.

```
pascal OSStatus GetThemeMenuBackgroundRegion (
                    const Rect *inMenuRect,
                    ThemeMenuType inMenuType,
                    RgnHandle Region);
```

`inMenuRect` On input, a pointer to a rectangle.

`inMenuType`	A value specifying the type of menu (pull-down, pop-up, or hierarchical) whose background you wish to obtain; see "Appearance-Compliant Menu Type Constants" (page 31).
`Region`	On input, a handle to a window region created by the application. On output, a handle to the region of the specified rectangle.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

The `GetThemeMenuBackgroundRegion` function should be called when you are writing an Appearance-compliant menu bar definition function. It gets the menu background region for the rectangle specified. This rectangle should be the entire menu. The region handle passed is set to the region.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `GetThemeMenuBackgroundRegion` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

GetThemeMenuBarHeight

NEW WITH THE APPEARANCE MANAGER

Gets the optimal height of a menu bar for the current theme.

```
pascal OSStatus GetThemeMenuBarHeight (SInt16 *outHeight);
```

`outHeight`	On output, the height (in pixels) of the menu bar.
function result	A result code; see "Result Codes" (page 33).

DISCUSSION

Call the `GetThemeMenuBarHeight` function if you are implementing a custom menu bar definition function and want to correctly calculate the height of a menu bar for the current theme. `GetThemeMenuBarHeight` will provide the ideal

height of a menu bar in the current theme and for this reason should be used instead of the `GetMBarHeight` function, which provides only the actual menu bar height. `GetMBarHeight` can provide misleading information if, for example, the menu bar is hidden; in that case `GetMBarHeight` would return 0 for the menu bar's height, while `GetThemeMenuBarHeight` would return the preferred height of the menu bar, whether or not it was currently drawn.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `GetThemeMenuBarHeight` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

GetThemeMenuSeparatorHeight

NEW WITH THE APPEARANCE MANAGER

Gets the height of a menu separator line for the current theme.

```
pascal OSStatus GetThemeMenuSeparatorHeight (SInt16 *outHeight);
```

`outHeight` On output, the height (in pixels) of the menu separator line.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `GetThemeMenuSeparatorHeight` function should be called when you are writing your own menu definition function and wish to calculate a menu rectangle for a separator to match the current theme.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `GetThemeMenuSeparatorHeight` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

GetThemeMenuItemExtra

NEW WITH THE APPEARANCE MANAGER

Gets of measurement of the space (in pixels) surrounding a menu item in the current theme.

```
pascal OSStatus GetThemeMenuItemExtra (
                    ThemeMenuItemType inItemType,
                    SInt16 *outHeight,
                    SInt16 *outWidth);
```

`inItemType` A constant of type `ThemeMenuItemType`, identifying the type of menu item for which you are interested in getting a measurement. See "Appearance-Compliant Menu Item Type Constants" (page 32).

`outHeight` On output, the value (in pixels) of the total amount of padding between the content of the menu item and the top and bottom of its frame. Your content's height plus the measurement provided by the `outHeight` parameter equals the total item height.

`outWidth` On output, the value (in pixels) of the total amount of padding between the content of the menu item and the left and right limits of the menu. Your content's width plus the measurement provided by the `outWidth` parameter equals the total item width.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `GetThemeMenuItemExtra` function should be called when you are writing your own menu definition function and wish to be Appearance-compliant. Once you have determined the height and width of the content of a menu item, call `GetThemeMenuItemExtra` to get a measurement in pixels of the space surrounding a menu item, including any necessary inter-item spacing, for the current theme. By combining the values for your menu item's content and the extra padding needed by the theme, you can derive the size of the rectangle needed to encompass both the content and the theme element together.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `GetThemeMenuItemExtra` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

GetThemeMenuTitleExtra

NEW WITH THE APPEARANCE MANAGER

Gets a measurement of the space (in pixels) surrounding a menu title in a given theme.

```
pascal OSStatus GetThemeMenuTitleExtra (
                    SInt16 *outWidth,
                    Boolean inIsSquished);
```

`outWidth` On output, the distance (in pixels) between the width of the menu title and its frame.

`inIsSquished` A Boolean value. If all the titles do not fit in the menu bar and you wish to condense the menu title's spacing to fit, set to `true`; if set to `false`, the menu title will not be condensed.

function result A result code; see "Result Codes" (page 33).

DISCUSSION

The `GetThemeMenuTitleExtra` function should be called when you are writing your own menu definition function and wish to be Appearance-compliant. Once you have determined the height and width of the content of a menu title, call `GetThemeMenuTitleExtra` to get the distance (in pixels) surrounding the menu title in the current theme. This includes space on either side of the menu title.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `GetThemeMenuTitleExtra` function. See "Appearance Manager Gestalt Selector

Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Coordinating Colors and Patterns With the Current Theme

SetThemeBackground

NEW WITH THE APPEARANCE MANAGER

Sets an element's background color or pattern to comply with the current theme.

```
pascal OSStatus SetThemeBackground (
                    ThemeBrush inBrush,
                    SInt16 inDepth,
                    Boolean inIsColorDevice);
```

`inBrush` A value representing the pattern or color to which the background is to be set; see "Appearance-Compliant Brush Type Constants" (page 23).

`inDepth` The bit depth (in pixels) of the current graphics port.

`inIsColorDevice`
A Boolean value. Set to `true` to indicate that you are drawing on a color device. Set to `false` for a monochrome device.

function result A result code. The result code `appearanceBadBrushIndexErr` indicates that the brush constant passed was not valid. For a list of other result codes, see "Result Codes" (page 33).

DISCUSSION

The `SetThemeBackground` function should be called each time you wish to draw an element in a specified brush constant using Appearance Manager draw functions.

The constant in the `inBrush` parameter can represent a color or pattern, depending on the current theme. Because it could specify a pattern, remember to save and restore the `pnPixPat` and `bkPixPat` fields of your graphics port when

saving the text and background colors. Because patterns in the `bkPixPat` field override the background color of the window, call the Window Manager function `BackPat` to set your background pattern to a normal white pattern. This will ensure that you can use `RGBBackColor` to set your background color to white, call the Window Manager function `EraseRect`, and get the expected results.

SetThemePen

NEW WITH THE APPEARANCE MANAGER

Sets an element's pen pattern or color to comply with the current theme.

```
pascal OSStatus SetThemePen (
                    ThemeBrush inBrush,
                    SInt16 inDepth,
                    Boolean inIsColorDevice);
```

`inBrush` A value representing the pattern or color to which the pen is to be set; see "Appearance-Compliant Brush Type Constants" (page 23).

`inDepth` The bit depth (in pixels) of the current graphics port.

`inIsColorDevice`
A Boolean value. Set to `true` to indicate that you are drawing on a color device. Set to `false` for a monochrome device.

function result A result code. The result code `appearanceBadBrushIndexErr` indicates that the brush constant passed in was not valid. For a list of other result codes, see "Result Codes" (page 33).

DISCUSSION

The `SetThemePen` function should be called each time you wish to draw an element in a specified brush constant using Appearance Manager draw functions.

The constant in the `inBrush` parameter can represent a color or pattern, depending on the current theme. Because it could specify a pattern, remember to save and restore the `pnPixPat` and `bkPixPat` fields of your graphics port when saving the text and background colors. Because patterns in the `pnPixPat` field

override the foreground color of the window, call the Window Manager function `PenPat` to set your foreground pattern to a normal white pattern. This will ensure that you can use `RGBForeColor` to set your foreground color to white, call the Window Manager function `PaintRect`, and get the expected results.

SetThemeTextColor

NEW WITH THE APPEARANCE MANAGER

Sets an element's foreground color for drawing text to comply with the current theme.

```
pascal OSStatus SetThemeTextColor (
                    ThemeTextColor inColor,
                    SInt16 inDepth,
                    Boolean inIsColorDevice);
```

`inColor` A value representing the color to which the foreground text is to be set; see "Appearance-Compliant Text Color Constants" (page 25).

`inDepth` The bit depth (in pixels) of the current graphics port.

`inIsColorDevice`
A Boolean value. Set to `true` to indicate that you are drawing on a color device. Set to `false` for a monochrome device.

function result A result code. The result code `appearanceBadTextColorIndexErr` indicates that the text color index passed was not valid. For a list of other result codes, see "Result Codes" (page 33).

DISCUSSION

The `SetThemeTextColor` function is typically used inside a `DeviceLoop` drawing procedure to set the foreground color for drawing text in order to coordinate with the current theme.

SetThemeWindowBackground

NEW WITH THE APPEARANCE MANAGER

Sets the Appearance-compliant color or pattern that the window background will be repainted to when `PaintOne` is called.

```
pascal OSStatus SetThemeWindowBackground (
                    WindowPtr inWindow,
                    ThemeBrush inBrush,
                    Boolean inUpdate);
```

`inWindow`	On input, a pointer to a window.
`inBrush`	A value representing the pattern or color to which the window background will be set; see "Appearance-Compliant Brush Type Constants" (page 23).
`inUpdate`	A Boolean value. If `true`, the content region of the window is to be invalidated and the window erased. If `false`, the window background is to be set but no drawing will occur on screen.
function result	A result code. The result code `appearanceBadBrushIndexErr` indicates that the brush constant passed was not valid. For a list of other result codes, see "Result Codes" (page 33).

DISCUSSION

The `SetThemeWindowBackground` function sets the color or pattern to which the Window Manager will erase the window background.

The constant in the `inBrush` parameter can represent a color or pattern, depending on the current theme. Because it could specify a pattern, remember to save and restore the `pnPixPat` and `bkPixPat` fields of your graphics port when saving the text and background colors. Because patterns in the `bkPixPat` field override the background color of the window, call the Window Manager function `BackPat` to set your background pattern to a normal white pattern. This will ensure that you can use `RGBBackColor` to set your background color to white, call the Window Manager function `EraseRect`, and get the expected results.

IsThemeInColor

NEW WITH THE APPEARANCE MANAGER

Checks to see whether the current theme would draw in color in the given environment.

```
pascal Boolean IsThemeInColor (
                    SInt16 inDepth,
                    Boolean inIsColorDevice);
```

`inDepth` The bit depth (in pixels) of the current graphics port.

`inIsColorDevice`
A Boolean value. Set to `true` to indicate that you are drawing on a color device. Set to `false` for a monochrome device.

function result Returns `true` if, given the depth and color device information, the theme would draw in color; returns `false`, if not.

DISCUSSION

The `IsThemeInColor` function is useful when you are drawing elements to match the current theme and need to determine whether or not the theme would be drawn in color or black and white. If the function returns `true`, you can draw in color; if it returns `false`, you should draw in black and white.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `IsThemeInColor` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

GetThemeAccentColors

NEW WITH THE APPEARANCE MANAGER

Gets a copy of the accent colors for the platinum theme.

```
pascal OSStatus GetThemeAccentColors (CTabHandle *outColors);
```

`outColors`	On output, a handle to the accent colors
function result	A result code; see "Result Codes" (page 33). The result `appearanceThemeHasNoAccents` is returned if the current theme has no accent colors.

DESCRIPTION

The `GetThemeAccentColors` function returns a copy of an element's accent colors, but only for the platinum theme. If `GetThemeAccentColors` is called when another theme is current, it returns an error.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling the `GetThemeAccentColors` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Defining Your Own Menu Drawing Callback Functions

This section describes application-defined callback functions supplied by the Appearance Manager for the creation of Appearance-compliant custom menu titles and menu items.

MyMenuTitleDrawingProc

NEW WITH THE APPEARANCE MANAGER

Draws a custom menu title that coordinates with the current theme.

The Appearance Manager declares the type for an application-defined menu title drawing function as follows:

```
typedef pascal void (*MenuTitleDrawingProcPtr)(const Rect *inBounds,
SInt16 inDepth, Boolean inIsColorDevice, SInt32 inUserData);
```

The Appearance Manager defines the data type `MenuTitleDrawingUPP` to identify the universal procedure pointer for an application-defined menu title drawing function:

```
typedef UniversalProcPtr MenuTitleDrawingUPP;
```

You typically use the `NewMenuTitleDrawingProc` macro like this:

```
MenuTitleDrawingUPP myMenuTitleDrawingUPP;
myMenuTitleDrawingUPP = NewMenuTitleDrawingProc(MyMenuTitleDrawingProc);
```

You typically use the `CallMenuTitleDrawingProc` macro like this:

```
CallMenuTitleDrawingProc(myMenuTitleDrawingUPP, inBounds, inDepth,
inIsColorDevice, inUserData);
```

Here's how to declare a custom menu title drawing function, if you were to name the function `MyMenuTitleDrawingProc`:

```
pascal void (MyMenuTitleDrawingProc)
                    (const Rect *inBounds,
                    SInt16 inDepth,
                    Boolean inIsColorDevice,
                    SInt32 inUserData);
```

`inBounds` On input, a pointer to a rectangle in which you should draw your menu title content.

`inDepth` The bit depth (in pixels) of the current graphics port.

`inIsColorDevice` A Boolean value. Set to `true` to indicate that you are drawing on a color device. Set to `false` for a monochrome device.

`inUserData` User data specifying how to draw the menu title content, passed in from the `inTitleData` parameter of `DrawThemeMenuTitle` (page 46).

DISCUSSION

Your menu title drawing function will be called clipped to the rectangle in which you are allowed to draw your content; do not draw outside this region. You should center your content vertically inside the content rectangle.

At the time your menu title drawing function is called, the foreground text color and mode is already set to draw in the specified state (enabled, selected,

disabled) and correct color for the theme. You do not need to set the color unless you have special drawing needs.

IMPORTANT

You should not depend on the background color for your menu title, so you should not call the `EraseRect` function from your menu title drawing function.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling your `MyMenuTitleDrawingProc` function. See "Appearance Manager Gestalt Selector Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

MyMenuItemDrawingProc

NEW WITH THE APPEARANCE MANAGER

Draws a custom menu item that coordinates with the current theme.

The Appearance Manager declares the type for an application-defined menu item drawing function as follows:

```
typedef pascal void (*MenuItemDrawingProcPtr)(const Rect *inBounds,
SInt16 inDepth, Boolean inIsColorDevice, SInt32 inUserData);
```

The Appearance Manager defines the data type `MenuItemDrawingUPP` to identify the universal procedure pointer for an application-defined menu item drawing function:

```
typedef UniversalProcPtr MenuItemDrawingUPP;
```

You typically use the `NewMenuItemDrawingProc` macro like this:

```
MenuItemDrawingUPP myMenuItemDrawingUPP;
myMenuItemDrawingUPP = NewMenuItemDrawingProc(MyMenuItemDrawingProc);
```

You typically use the `CallMenuItemDrawingProc` macro like this:

```
CallMenuItemDrawingProc(myMenuItemDrawingUPP, inBounds, inDepth,
inIsColorDevice, inUserData);
```

Here's how to declare a custom menu item drawing function, if you were to name the function `MyMenuItemDrawingProc`:

```
pascal void (MyMenuItemDrawingProc)
                    (const Rect *inBounds,
                    SInt16 inDepth,
                    Boolean inIsColorDevice,
                    SInt32 inUserData);
```

`inBounds` On input, a pointer to a rectangle in which you should draw your menu item content.

`inDepth` The bit depth (in pixels) of the current graphics port.

`inIsColorDevice` A Boolean value. Set to `true` to indicate that you are drawing on a color device. Set to `false` for a monochrome device.

`inUserData` User data specifying how to draw the menu item content, passed in from the `inUserData` parameter of `DrawThemeMenuItem` (page 48).

DISCUSSION

Your menu item drawing function will be called clipped to the rectangle in which you are allowed to draw your content; do not draw outside this region. You should center your content vertically inside the content rectangle.

At the time your menu item drawing function is called, the foreground text color and mode is already set to draw in the specified state (enabled, selected, disabled) and correct color for the theme. You do not need to set the color unless you have special drawing needs.

IMPORTANT

You should not depend on the background color for your menu item, so you should not call the `EraseRect` function from your menu item drawing function.

SPECIAL CONSIDERATIONS

Make sure Appearance Manager 1.0.1 is present before calling your `MyMenuItemDrawingProc` function. See "Appearance Manager Gestalt Selector

Constants" (page 21) for details on how to determine if the Appearance Manager is present and what its version is, if so.

Control Manager Reference

Contents

This chapter describes the Control Manager types and constants, resources, and functions that are affected by Mac OS 8 or the Appearance Manager.

- "Control Manager Types and Constants" (page 71) describes Control Manager types and constants, including structures. Result codes are included at the end of this section.
- "Control Manager Resources" (page 111) describes the control (`'CNTL'`) resource, the control color table (`'cctb'`) resource, the list description (`'ldes'`) resource, the tab information (`'tab#'`) resource, and the control definition function (`'CDEF'`) resource.
- "Control Manager Functions" (page 118) describes both Control Manager functions and application-defined callback functions.

Control Manager Types and Constants

Control Definition IDs

When creating a control, your application supplies a control definition ID to the control resource (page 111) or to one of the Control Manager control-creation functions. The control definition ID indicates the type of control to create. A **control definition ID** is an integer that contains the resource ID of a control definition function in its upper 12 bits and a variation code in its lower 4 bits. A control definition ID is derived as follows:

control definition ID = 16 * (`'CDEF'` resource ID) + variation code

A **control definition function** determines how a control generally looks and behaves. Control definition functions are stored as resources of type `'CDEF'`. Various Control Manager functions call a control definition function whenever they need to perform some control-dependent action, such as drawing the control on the screen. For more information on how to create a control definition function, see "Defining Your Own Control Definition Function" (page 162).

A control definition function, in turn, can use a **variation code** to describe variations of the same basic control. For example, all pop-up arrows share the same basic control definition function, which is stored in a resource of type

'CDEF' and has a resource ID of 12. The standard pop-up arrow is large and points to the right; it has a control definition ID of 192. A variation of this is a large, left-pointing arrow, which has a control definition ID of 193. Still another variation, in which the arrow points up, has a control definition ID of 194.

Your application can use the constants listed in Table 2-1 in place of control definition IDs. Most of these constants, and their associated IDs, are new with the Appearance Manager and are not supported unless the Appearance Manager is available. A control definition ID that is new is identified with an asterisk (*) in its description in Table 2-1. For illustrations of these new controls, see "Control Guidelines" in *Mac OS 8 Human Interface Guidelines*.

If your application contains code that uses the older, pre-Appearance control definition IDs or their constants, your application can use the Appearance Manager to map the old IDs to those for the new, updated controls introduced by the Appearance Manager. In particular, the control definition IDs for pre-Appearance checkboxes, buttons, scroll bars, radio buttons, and pop-up menus will be automatically mapped to Appearance-compliant equivalents. For more information about this mapping, see "Introduction to the Appearance Manager" (page 19).

Table 2-1 Control definition IDs and resource IDs for standard controls

Constant (and Value) for Control Definition ID	Description	Resource ID
pushButProc (0)	Pre-Appearance push button.	0
pushButProc + kControlUsesOwningWindowsFontVariant (8)	Pre-Appearance push button with its text in the window font.	0
kControlPushButtonProc (368)	Appearance-compliant push button.*	23
kControlPushButLeftIconProc (374)	Appearance-compliant push button with a color icon to the left of the control title.* (This direction is reversed when the system justification is right to left). The contrlMax field of the control structure for this control contains the resource ID of the 'cicn' resource drawn in the pushbutton.	23

Table 2-1 Control definition IDs and resource IDs for standard controls (continued)

Constant (and Value) for Control Definition ID	Description	Resource ID
kControlPushButRightIconProc (375)	Appearance-compliant push button with a color icon to right of control title.* (This direction is reversed when the system justification is right to left). The contrlMax field of the control structure for this control contains the resource ID of the 'cicn' resource drawn in the pushbutton.	23
checkBoxProc (1)	Pre-Appearance checkbox.	0
checkBoxProc + kControlUsesOwningWindowsFontVariant (8)	Pre-Appearance checkbox with a control title in the window font.	0
kControlCheckBoxProc (369)	Appearance-compliant checkbox.*	23
radioButProc (2)	Pre-Appearance radio button.	0
radioButProc + kControlUsesOwningWindowsFontVariant (8)	Pre-Appearance radio button with a title in the window font.	0
kControlRadioButtonProc (370)	Appearance-compliant radio button.*	23
scrollBarProc (16)	Pre-Appearance scroll bar.	1
kControlScrollBarProc (384)	Appearance-compliant scroll bar.*	24
kControlScrollBarLiveProc (386)	Appearance-compliant scroll bar with live feedback.*	24
kControlBevelButtonSmallBevelProc (32)	Bevel button with a small bevel.*	2
kControlBevelButtonNormalBevelProc (33)	Bevel button with a normal bevel.*	2
kControlBevelButtonLargeBevelProc (34)	Bevel button with a large bevel.*	2
kControlBevelButtonSmallBevelProc + kControlBevelButtonMenuOnRight (4)	Small bevel button with a pop-up menu.*	2
kControlSliderProc (48)	Slider.* Your application calls the function SetControlAction (page 153) to set the last value for the control.	3

Table 2-1 Control definition IDs and resource IDs for standard controls (continued)

Constant (and Value) for Control Definition ID	Description	Resource ID
`kControlSliderProc` + `kControlSliderLiveFeedback` (1)	Slider with live feedback.* The value of the control is updated automatically by the Control Manager before your action function is called. If no application-defined action function is supplied, the slider draws an outline of the indicator as the user moves it.	3
`kControlSliderProc` + `kControlSliderHasTickMarks` (2)	Slider with tick marks.* The control rectangle must be large enough to include the tick marks.	3
`kControlSliderProc` + `kControlSliderReverseDirection` (4)	Slider with a directional indicator.* The indicator is positioned perpendicularly to the slider; that is, if the slider is horizontal, the indicator points up, and if the slider is vertical, the indicator points left.	3
`kControlSliderProc` + `kControlSliderNonDirectional` (8)	Slider with a rectangular, non-directional indicator.* This variant overrides the `kSliderReverseDirection` and `kSliderHasTickMarks` variants.	3
`kControlTriangleProc` (64)	Disclosure triangle.*	4
`kControlTriangleLeftFacingProc` (65)	Left-facing disclosure triangle.*	4
`kControlTriangleAutoToggleProc` (66)	Auto-tracking disclosure triangle.*	4
`kControlTriangleLeftFacingAutoToggleProc` (67)	Left-facing, auto-tracking disclosure triangle.*	4
`kControlProgressBarProc` (80)	Progress indicator.* To make the control determinate or indeterminate, set the `kControlProgressBarIndeterminateTag` constant; see "Control Data Tag Constants" (page 83). Progress indicators are only horizontal in orientation; vertical progress indicators are not currently supported.	5

Table 2-1 Control definition IDs and resource IDs for standard controls (continued)

Constant (and Value) for Control Definition ID	Description	Resource ID
kControlLittleArrowsProc (96)	Little arrows.*	6
kControlChasingArrowsProc (11)	Asynchronous arrows.*	7
kControlTabLargeProc (128)	Normal tab control.*	8
kControlTabSmallProc (129)	Small tab control.*	
kControlSeparatorLineProc (144)	Separator line.	9
kControlGroupBoxTextTitleProc (160)	Primary group box with text title.*	10
kControlGroupBoxCheckBoxProc (161)	Primary group box with checkbox title.*	10
kControlGroupBoxPopupButtonProc (162)	Primary group box with pop-up button title.*	10
kControlGroupBoxSecondaryTextTitleProc (164)	Secondary group box with text title.*	10
kControlGroupBoxSecondaryCheckBoxProc (165)	Secondary group box with checkbox title.*	10
kControlGroupBoxSecondaryPopupButtonProc (166)	Secondary group box with pop-up button title.*	10
kControlImageWellProc (176)	Image well.* This control behaves as a palette-type object: it can be selected by clicking, and clicking on another object should change the keyboard focus. If the keyboard focus is removed, your application should then set the value to 0 to remove the checked border.	11
kControlImageWellAutoTrackProc (177)	Image well with autotracking.* This variant sets the value itself so the control remains highlighted.	11
kControlPopupArrowEastProc (192)	Large, right-facing pop-up arrow.*	12
kControlPopupArrowWestProc (193)	Large, left-facing pop-up arrow.*	12

Table 2-1 Control definition IDs and resource IDs for standard controls (continued)

Constant (and Value) for Control Definition ID	Description	Resource ID
kControlPopupArrowNorthProc (194)	Large, up-facing pop-up arrow.*	12
kControlPopupArrowSouthProc (195)	Large, down-facing pop-up arrow.*	12
kControlPopupArrowSmallEastProc (196)	Small, right-facing pop-up arrow.*	12
kControlPopupArrowSmallWestProc (197)	Small, left-facing pop-up arrow.*	12
kControlPopupArrowSmallNorthProc (198)	Small, up-facing pop-up arrow.*	12
kControlPopupArrowSmallSouthProc (199)	Small, down-facing pop-up arrow.*	12
kControlPlacardProc (224)	Placard.*	14
kControlClockTimeProc (240)	Clock control displaying hour/minutes.*	15
kControlClockTimeSecondsProc (241)	Clock control displaying hours/minutes/seconds.*	15
kControlClockDateProc (242)	Clock control displaying date/month/year.*	15
kControlClockMonthYearProc (243)	Clock control displaying month/year.*	15
kControlUserPaneProc (256)	User pane.*	16
kControlEditTextProc (272)	Editable text field for windows.* This control maintains its own text handle (TEHandle).	17
kControlEditTextDialogProc (273)	Editable text field for dialog boxes.* This control uses the dialog box common text handle.	17
kControlEditTextPasswordProc (274)	Editable text field for passwords.* This control is supported by the Script Manager. Password text can be accessed via the kEditTextPasswordTag constant; see "Control Data Tag Constants" (page 83).	17
kControlStaticTextProc (288)	Static text field.*	18
kControlPictureProc (304)	Picture control.*	19

Table 2-1 Control definition IDs and resource IDs for standard controls (continued)

Constant (and Value) for Control Definition ID	Description	Resource ID
`kControlPictureNoTrackProc` (305)	Non-tracking picture.* Immediately returns `kControlPicturePart` as the part code hit without tracking.	19
`kControlIconProc` (320)	Icon control.*	20
`kControlIconNoTrackProc` (321)	Non-tracking icon.*	20
`kControlIconSuiteProc` (322)	Icon suite.*	20
`kControlIconSuiteNoTrackProc` (323)	Non-tracking icon suite.*	20
`kControlWindowHeaderProc` (336)	Window header.*	21
`kControlWindowListViewHeaderProc` (337)	Window list view header.*	21
`kControlListBoxProc` (352)	List box.*	21
`kControlListBoxAutoSizeProc` (353)	Autosizing list box.*	21
`popupMenuProc` (1008)	Pre-Appearance standard pop-up menu.	63
`popupMenuProc + popupFixedWidth` (1009)	Pre-Appearance, fixed-width pop-up menu.	63
`popupMenuProc + popupVariableWidth` (1010)	Pre-Appearance, variable-width pop-up menu.	63
`popupMenuProc + popupUseAddResMenu` (1012)	Pre-Appearance pop-up menu with a value of type `ResType` in the `contrlRfCon` field of the control structure. The Menu Manager adds resources of this type to the menu.	63
`popupMenuProc + popupUseWFont` (1016)	Pre-Appearance pop-up menu with a control title in the window font.	63
`kControlPopupButtonProc` (400)	Appearance-compliant standard pop-up menu.*	25
`kControlPopupButtonProc + kControlPopupFixedWidthVariant` (1)	Appearance-compliant fixed-width pop-up menu.*	25
`kControlPopupButtonProc + kControlPopupVariableWidthVariant` (2)	Appearance-compliant variable-width pop-up menu.*	25

Table 2-1 Control definition IDs and resource IDs for standard controls (continued)

Constant (and Value) for Control Definition ID	Description	Resource ID
`kControlPopupButtonProc + kControlPopupUseAddResMenuVariant` (4)	Appearance-compliant pop-up menu with a value of type `ResType` in the `contrlRfCon` field of the control structure.* The Menu Manager adds resources of this type to the menu.	25
`kControlPopupButtonProc + kControlPopupUseWFontVariant` (8)	Appearance-compliant pop-up menu with control title in window font.*	25
`kControlRadioGroupProc` (416)	Radio group.* Embedder control for controls that have set the feature bit `kControlHasRadioBehavior`.	26

* This control definition is new with the Appearance Manager and is not supported unless the Appearance Manager is available.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Only the following control definition IDs are supported:

```
enum {
    pushButProc     = 0,
    checkBoxProc    = 1,
    radioButProc    = 2,
    useWFont        = 8,
    scrollBarProc   = 16,
    popupMenuProc   = 1008,
/*pop-up menu CDEF variation codes*/
    popupFixedWidth     = 1 << 0,
    popupVariableWidth  = 1 << 1,
    popupUseAddResMenu  = 1 << 2,
    popupUseWFont       = 1 << 3
};
```

Settings Values for Standard Controls

This section lists the initial, minimum, and maximum settings for all standard controls. You should set these values in the control (`'CNTL'`) resource (page 111)

when creating a new control from a resource or pass these values in the `minimumValue`, `maximumValue`, and `initialValue` parameters of `NewControl` (page 119).

Some controls specify other information besides their range in their minimum and maximum settings. For example, bevel buttons use the high byte of their minimum value to indicate their behavior.

Control Values

Push button (pre-Appearance)
Initial: 0
Minimum: 0
Maximum: 1

Push button (Appearance-compliant)
Initial: 0
Minimum: 0
Maximum: 1

Checkbox (pre-Appearance)
Initial: `kControlCheckboxUncheckedValue`
Minimum: `kControlCheckboxUncheckedValue`
Maximum: `kControlCheckboxCheckedValue`

Checkbox (Appearance-compliant)
Initial: `kControlCheckboxUncheckedValue`
Minimum: `kControlCheckboxUncheckedValue`
Maximum: `kControlCheckboxCheckedValue` or `kControlCheckboxMixedValue`

Radio button (pre-Appearance)
Initial: `kControlRadioButtonUncheckedValue`
Minimum: `kControlRadioButtonUncheckedValue`
Maximum: `kControlRadioButtonCheckedValue`

Radio button (Appearance-compliant)
Initial: `kControlRadioButtonUncheckedValue`
Minimum: `kControlRadioButtonUncheckedValue`
Maximum: `kControlRadioButtonCheckedValue` or `kControlRadioButtonMixedValue`

Scroll bar (pre-Appearance and Appearance-compliant versions)
Initial: Appropriate value between –32768 and 32768.
Minimum: –32768 to 32768

	Maximum: –32768 to 32768; when the maximum setting is equal to the minimum setting, the scroll bar is inactive.
Bevel button	**Initial:** If you wish to attach a menu, the menu ID; if no menu is attached, 0. **Minimum:** High byte specifies behavior—see "Bevel Button Behavior Constants" (page 92). Low byte specifies content type—see "Bevel Button and Image Well Content Type Constants" (page 94). **Maximum:** Resource ID of bevel button's content if resource-based—see "Bevel Button and Image Well Content Type Constants" (page 94).
Slider	**Initial:** Appropriate value between –32768 and 32768; for tick mark variant, the number of ticks; reset to the minimum setting after creation. **Minimum:** –32768 to 32768 **Maximum:** –32768 to 32768; when the maximum setting is equal to the minimum setting, the slider is inactive.
Disclosure triangle	**Initial:** 0 (collapsed) or 1 (expanded) **Minimum:** 0 (collapsed) **Maximum:** 1 (expanded)
Progress indicator	**Initial:** Appropriate value between –32768 and 32768. **Minimum:** –32768 to 32768 **Maximum:** –32768 to 32768
Little arrows	**Initial:** Appropriate value between –32768 and 32768. **Minimum:** –32768 to 32768 **Maximum:** –32768 to 32768
Asynchronous arrows	**Initial:** Reserved. Set to 0. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Tab control	**Initial:** Resource ID of the `'tab#'` resource you are using to hold tab information. Reset to the minimum setting after creation. Under Appearance 1.0.1 and later, a value of 0 indicates not to read a `'tab#'` resource; see "The Tab Information Structure" (page 109). **Minimum:** Ignored. Reset to 1 after creation. **Maximum:** Ignored. Reset to the number of individual tabs in the tab control after creation.

Separator line	**Initial:** Reserved. Set to 0. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Primary group box and secondary group box	**Initial:** Ignored if group box has text title. If the group box has a checkbox or pop-up button title, same value as the checkbox or pop-up button. **Minimum:** Ignored if group box has text title. If the group box has a checkbox or pop-up button title, same minimum setting as the checkbox or pop-up button. **Maximum:** Ignored if group box has text title. If the group box has a checkbox or pop-up button title, same maximum setting as the checkbox or pop-up button.
Image well	**Initial:** If you wish to attach a menu, the menu ID. If no menu is attached, 0. Resource ID of bevel button's content if resource-based—see "Bevel Button and Image Well Content Type Constants" (page 94). Reset to 0 after creation. **Minimum:** High byte specifies behavior—see "Bevel Button Behavior Constants" (page 92). Low byte specifies content type—see "Bevel Button and Image Well Content Type Constants" (page 94). After the image well is created, reset to 0. **Maximum:** Ignored. Reset to 2 after creation.
Pop-up arrow	**Initial:** Reserved. Set to 0. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Placard	**Initial:** Reserved. Set to 0. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Clock	**Initial:** One or more of the clock value flags—see "Clock Value Flag Constants" (page 98). Reset to 0 after creation. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
User pane	**Initial:** One or more of the control feature constants—see "Specifying Which Appearance-Compliant Messages Are Supported" (page 174). Reset to 0 after creation. **Minimum:** Ignored. After user pane creation, reset to a setting between –32768 to 32768.

	Maximum: Ignored. Reset to a setting between –32768 to 32768 after creation.
Editable text field	**Initial:** Reserved. Set to 0. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Static text field	**Initial:** Reserved. Set to 0. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Picture	**Initial:** Resource ID of the `'pict'` resource you wish to display; reset to 0 after creation. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Icon	**Initial:** Resource ID of the `'cicn'`, `'ICON'`, or icon suite resource you wish to display. For icon suite variant, it only looks for an icon suite. If not, it looks for a `'cicn'` or `'ICON'` resource. Reset to 0 after creation. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Window header	**Initial:** Reserved. Set to 0. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
List box	**Initial:** Resource ID of the `'ldes'` resource you are using to hold list box information; reset to 0 after creation. An initial value of 0 indicates not to read an `'ldes'` resource under Appearance 1.0.1 and later. **Minimum:** Reserved. Set to 0. **Maximum:** Reserved. Set to 0.
Pop-up menu (pre-Appearance and Appearance-compliant versions)	**Initial:** One or more of the pop-up menu title constants. **Minimum:** Resource ID of the `'MENU'` resource. **Maximum:** Width (in pixels) of the pop-up menu title.
Radio group	**Initial:** Set to 0 on creation. Reset to the index of currently selected embedded radio control after creation. If currently selected control does not support radio behavior, value will be set to 0 and the control will be deselected. To deselect all controls, set to 0. **Minimum:** Set to 0. **Maximum:** Set to 0 on creation. Reset to the number of embedded controls as controls are added.

Control Data Tag Constants

NEW WITH THE APPEARANCE MANAGER

The constants described here are passed in the `inTagName` parameters of `SetControlData` (page 154) and `GetControlData` (page 156) to specify the piece of data in a control that you wish to set or get. You can also pass these constants in the `inTagName` parameter of `GetControlDataSize` (page 157) if you wish to determine the size of variable-length control data (e.g., text in an editable text control). These constants can also be used by custom control definition functions that return the feature bit `kControlSupportsDataAccess` in response to a `kControlMsgGetFeatures` message.

The data that your application gets or sets can be of various types. The list below shows the data types for the information that you can set in the `inData` parameter to the `SetControlData` function and that you can get in the `inBuffer` parameter to the `GetControlData` function.

```
enum {
    kControlPushButtonDefaultTag            = ('dflt'),
    kControlBevelButtonContentTag           = ('cont'),
    kControlBevelButtonTransformTag         = ('tran'),
    kControlBevelButtonTextAlignTag         = ('tali'),
    kControlBevelButtonTextOffsetTag        = ('toff'),
    kControlBevelButtonGraphicAlignTag      = ('gali'),
    kControlBevelButtonGraphicOffsetTag     = ('goff'),
    kControlBevelButtonTextPlaceTag         = ('tplc'),
    kControlBevelButtonMenuValueTag         = ('mval'),
    kControlBevelButtonMenuHandleTag        = ('mhnd'),
    kControlBevelButtonCenterPopupGlyphTag  = ('pglc'),
    kControlTriangleLastValueTag            = ('last'),
    kControlProgressBarIndeterminateTag     = ('inde'),
    kControlTabContentRectTag               = ('rect'),
    kControlTabEnabledFlagTag               = ('enab'),
    kControlTabInfoTag                      = ('tabi'),
    kControlGroupBoxMenuHandleTag           = ('mhan'),
    kControlImageWellContentTag             = ('cont'),
    kControlImageWellTransformTag           = ('tran'),
    kControlClockLongDateTag                = ('date'),
    kControlUserItemDrawProcTag             = ('uidp'),
```

```
    kControlUserPaneDrawProcTag             = ('draw'),
    kControlUserPaneHitTestProcTag          = ('hitt'),
    kControlUserPaneTrackingProcTag         = ('trak'),
    kControlUserPaneIdleProcTag             = ('idle'),
    kControlUserPaneKeyDownProcTag          = ('keyd'),
    kControlUserPaneActivateProcTag         = ('acti'),
    kControlUserPaneFocusProcTag            = ('foci'),
    kControlUserPaneBackgroundProcTag       = ('back'),
    kControlEditTextTextTag                 = ('text'),
    kControlEditTextTEHandleTag             = ('than'),
    kControlEditTextSelectionTag            = ('sele'),
    kControlEditTextPasswordTag             = ('pass'),
    kControlStaticTextTextTag               = ('text'),
    kControlStaticTextTextHeightTag         = ('thei'),
    kControlIconTransformTag                = ('trfm'),
    kControlIconAlignmentTag                = ('algn'),
    kControlListBoxListHandleTag            = ('lhan'),
    kControlFontStyleTag                    = ('font'),
    kControlKeyFilterTag                    = ('fltr'),
    kControlBevelButtonLastMenuTag          = ('lmnu'),
    kControlBevelButtonMenuDelayTag         = ('mdly'),
    kControlPopupButtonMenuHandleTag        = ('mhan'),
    kControlPopupButtonMenuIDTag            = ('mnid'),
    kControlListBoxDoubleClickTag           = ('dblc'),
    kControlListBoxLDEFTag                  = ('ldef')
};
```

Constant descriptions

`kControlPushButtonDefaultTag`

Tells Appearance-compliant button whether to draw a default ring, or returns whether the Appearance Manager draws a default ring for the button.
Data type returned or set: `Boolean`

`kControlBevelButtonContentTag`

Gets or sets a bevel button's content type for drawing; see "Bevel Button and Image Well Content Type Constants" (page 94).
Data type returned or set: `ControlButtonContentInfo` structure

`kControlBevelButtonCenterPopUpGlyphTag`

Gets or sets the position of the pop-up arrow in a bevel button when a pop-up menu is attached.
Data type returned or set: `Boolean`; if `true`, glyph is vertically centered on the right; if `false`, glyph is on the bottom right.

`kControlBevelButtonTransformTag`

Gets or sets a transform that is added to the standard transform of a bevel button; see "Icon Utilities" in *More Macintosh Toolbox*.
Data type returned or set: `IconTransformType`

`kControlBevelButtonTextAlignTag`

Gets or sets the alignment of text in a bevel button; see "Bevel Button Text Alignment Constants" (page 97).
Data type returned or set: `ControlButtonTextAlignment`

`kControlBevelButtonTextOffsetTag`

Gets or sets the number of pixels that text is offset in a bevel button from the button's left or right edge; this is used with left, right, or system justification, but it is ignored when the text is center aligned.
Data type returned or set: `SInt16`

`kControlBevelButtonGraphicAlignTag`

Gets or sets the alignment of graphics in a bevel button in relation to any text the button may contain; see "Bevel Button Graphic Alignment Constants" (page 95).
Data type returned or set: `ControlButtonGraphicAlignment`

`kControlBevelButtonGraphicOffsetTag`

Gets or sets the horizontal and vertical amounts that a graphic element contained in a bevel button is offset from the button's edges; this value is ignored when the graphic is specified to be centered on the button.
Data type returned or set: `point`

`kControlBevelButtonTextPlaceTag`

Gets or sets the placement of a bevel button's text; see "Bevel Button Text Placement Constants" (page 97).
Data type returned or set: `ControlButtonTextPlacement`

`kControlBevelButtonMenuValueTag`

Gets the menu value for a bevel button with an attached menu; see "Bevel Button Menu Constants" (page 93).
Data type returned: `SInt16`

`kControlBevelButtonMenuHandleTag`

Gets the menu handle for a bevel button with an attached menu.
Data type returned: `MenuHandle`

`kControlTriangleLastValueTag`

Gets or sets the last value of a disclosure triangle. Used primarily for setting up a disclosure triangle properly when using the auto-toggle variant.
Data type returned or set: `SInt16`

`kControlProgressBarIndeterminateTag`

Gets or sets whether a progress indicator is determinate or indeterminate.
Data type returned or set: `Boolean`; if `true`, switches to an indeterminate progress indicator; if `false`, switches to an determinate progress indicator.

`kControlTabContentRectTag`

Gets the content rectangle of a tab control.
Data type returned: `Rect`

`kControlTabEnabledFlagTag`

Enables or disables a single tab in a tab control.
Data type returned or set: `Boolean`; if `true`, enabled; if `false`, disabled.

`kControlTabInfoTag`

Gets or sets information for a tab in a tab control; see "The Tab Information Structure" (page 109). Available with Appearance 1.0.1 and later.
Data type returned or set: `ControlTabInfoRec`.

`kControlGroupBoxMenuHandleTag`

Gets the menu handle of a group box.
Data type returned: `MenuHandle`

`kControlImageWellContentTag`

Gets or sets the content for an image well; see "The Bevel Button and Image Well Content Structure" (page 107).

Data type returned or set: `ControlButtonContentInfo` structure

`kControlImageWellTransformTag`

Gets or sets a transform that is added to the standard transform of an image well; see "Icon Utilities" in *More Macintosh Toolbox*.
Data type returned or set: `IconTransformType`

`kControlClockLongDateTag`

Gets or sets the clock control's time or date.
Data type returned or set: `LongDateRec` structure

`kControlUserItemDrawProcTag`

Gets or sets an application-defined item drawing function. If an embedding hierarchy is established, a user pane drawing function should be used instead of an item drawing function.
Data type returned or set: `UserItemUPP`

`kControlUserPaneDrawProcTag`

Gets or sets a user pane drawing function; see `MyUserPaneDrawProc` (page 189). Indicates that the Control Manager needs to draw a control.
Data type returned or set: `ControlUserPaneDrawingUPP`

`kControlUserPaneHitTestProcTag`

Gets or sets a user pane hit-testing function. Indicates that the Control Manager needs to determine if a control part was hit; see `MyUserPaneHitTestProc` (page 190).
Data type returned or set: `ControlUserPaneHitTestUPP`

`kControlUserPaneTrackingProcTag`

Gets or sets a user pane tracking function, which will be called when a control definition function returns the `kControlHandlesTracking` feature bit in response to a `kControlMsgGetFeatures` message. Indicates that a user pane handles its own tracking; see `MyUserPaneTrackingProc` (page 191).
Data type returned or set: `ControlUserPaneTrackingUPP`

`kControlUserPaneIdleProcTag`

Gets or sets a user pane idle function, which will be called when a control definition function returns the `kControlWantsIdle` feature bit in response to a

kControlMsgGetFeatures message. Indicates that a user pane performs idle processing; see MyUserPaneIdleProc (page 193).
Data type returned or set: ControlUserPaneIdleUPP

kControlUserPaneKeyDownProcTag
Gets or sets a user pane key down function, which will be called when a control definition function returns the kControlSupportsFocus feature bit in response to a kControlMsgGetFeatures message. Indicates that a user pane performs keyboard event processing; see MyUserPaneKeyDownProc (page 194).
Data type returned or set: ControlUserPaneKeyDownUPP

kControlUserPaneActivateProcTag
Gets or sets a user pane activate function, which will be called when a control definition function returns the kControlWantsActivate feature bit in response to a kControlMsgGetFeatures message. Indicates that a user pane wants to be informed of activate and deactivate events; see MyUserPaneActivateProc (page 195).
Data type returned or set: ControlUserPaneActivateUPP

kControlUserPaneFocusProcTag
Gets or sets a user pane keyboard focus function, which will be called when a control definition function returns the kControlSupportsFocus feature bit in response to a kControlMsgGetFeatures message. Indicates that a user pane handles keyboard focus; see MyUserPaneFocusProc (page 197).
Data type returned or set: ControlUserPaneFocusUPP

kControlUserPaneBackgroundProcTag
Gets or sets a user pane background function, which will be called when a control definition function returns the kControlHasSpecialBackground and kControlSupportsEmbedding feature bits in response to a kControlMsgGetFeatures message. Indicates that a user pane can set its background color or pattern; see MyUserPaneBackgroundProc (page 198).
Data type returned or set: ControlUserPaneBackgroundUPP

kControlEditTextTextTag
Gets or sets text in an editable text control.
Data type returned or set: character buffer

`kControlEditTextTEHandleTag`
Gets a handle to a text edit structure.
Data type returned: `TEHandle`

`kControlEditTextSelectionTag`
Gets or sets the selection in an editable text control.
Data type returned or set: `ControlEditTextSelectionRec` structure

`kControlEditTextPasswordTag`
Gets clear password text from an editable text control, that is, the text of the actual password typed, not the bullet text.
Data type returned: character buffer

`kControlStaticTextTextTag`
Gets or sets text in a static text control.
Data type returned or set: character buffer

`kControlStaticTextTextHeightTag`
Gets the height of text in a static text control.
Data type returned or set: `SInt16`

`kControlIconTransformTag`
Gets or sets a transform that is added to the standard transform of an icon; see "Icon Utilities" in *More Macintosh Toolbox*.
Data type returned or set: `IconTransformType`

`kControlIconAlignmentTag`
Gets or sets an icon's position (centered, left, right); see "Icon Utilities" in *More Macintosh Toolbox*.
Data type returned or set: `IconAlignmentType`

`kControlListBoxListHandleTag`
Gets a handle to a list box.
Data type returned: `ListHandle`

`kControlFontStyleTag`
Gets or sets the font style for controls that support text (includes list box, tab, clock, static and editable text).
Data type returned or set: `kControlFontStyleTag`

`kControlKeyFilterTag`
Gets or sets the key filter function for controls that handle filtered input (includes editable text and list box).
Data type returned or set: `ControlKeyFilterUPP`

`kControlBevelButtonLastMenuTag`

Gets the menu ID of the last menu selected in the submenu or main menu. Available with Appearance 1.0.1 and later.
Data type returned: `SInt16`

`kControlBevelButtonMenuDelayTag`

Gets or sets the delay (in number of ticks) before the menu is displayed. Available with Appearance 1.0.1 and later.
Data type returned or set: `SInt32`

`kControlPopupButtonMenuHandleTag`

Gets or sets the menu handle for a pop-up menu. Available with Appearance 1.0.1 and later.
Data type returned or set: `MenuHandle`

`kControlPopupButtonMenuIDTag`

Gets or sets the menu ID for a pop-up menu. Available with Appearance 1.0.1 and later.
Data type returned or set: `SInt16`

`kControlListBoxDoubleClickTag`

Checks to see whether the most recent click in a list box was a double click. Available with Appearance 1.0.1 and later.
Data type returned: `Boolean`; if `true`, the last click was a double click; if `false`, not.

`kControlListBoxLDEFTag`

Sets the `'LDEF'` resource to be used to draw a list box's contents; this is useful for creating a list box without an `'ldes'` resource. Available with Appearance 1.0.1 and later.
Data type set: `SInt16`.

Checkbox Value Constants

CHANGED WITH THE APPEARANCE MANAGER

These constants specify the value of a standard checkbox control and are passed in the `newValue` parameter of `SetControlValue` and are returned by `GetControlValue`.

```
enum {
    kControlCheckboxUncheckedValue  = 0,
    kControlCheckboxCheckedValue    = 1,
    kControlCheckboxMixedValue      = 2
};
```

Constant descriptions

`kControlCheckboxUncheckedValue`
The checkbox is unchecked.

`kControlCheckboxCheckedValue`
The checkbox is checked.

`kControlCheckboxMixedValue`
Mixed value. Indicates that a setting is on for some elements in a selection and off for others. This state only applies to standard Appearance-compliant checkboxes.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Pre-Appearance checkboxes do not support the mixed state value constant `kControlCheckboxMixedValue`.

Radio Button Value Constants

CHANGED WITH THE APPEARANCE MANAGER

These constants specify the value of a standard radio button control and are passed in the `newValue` parameter of `SetControlValue` and are returned by `GetControlValue`.

```
enum {
    kControlRadioButtonUncheckedValue   = 0,
    kControlRadioButtonCheckedValue     = 1,
    kControlRadioButtonMixedValue       = 2
};
```

Constant descriptions

`kControlRadioButtonUncheckedValue`
The radio button is unselected.

`kControlRadioButtonCheckedValue`
The radio button is selected.

`kControlRadioButtonMixedValue`
Mixed value. Indicates that a setting is on for some elements in a selection and off for others. This state only applies to standard Appearance-compliant radio buttons.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Pre-Appearance radio buttons do not support the mixed state value constant `kControlRadioButtonMixedValue`.

Bevel Button Behavior Constants

NEW WITH THE APPEARANCE MANAGER

You can pass these constants in the high byte of the `minimumValue` parameter of `NewControl` (page 119) to create a bevel button with a specific behavior.

```
enum {
    kControlBehaviorPushbutton      = 0,
    kControlBehaviorToggles         = 0x0100,
    kControlBehaviorSticky          = 0x0200,
    kControlBehaviorOffsetContents  = 0x8000
};
```

Constant descriptions

`kControlBehaviorPushbutton`
Push button (momentary) behavior. The bevel button pops up after being clicked.

`kControlBehaviorToggles`
Toggle behavior. The bevel button toggles state automatically when clicked.

`kControlBehaviorSticky`
Sticky behavior. Once clicked, the bevel button stays down until your application sets the control's value to 0. This behavior is useful in tool palettes and radio groups.

`kControlBehaviorOffsetContents`
: Bevel button contents are offset (one pixel down and to the right) when button is pressed.

Bevel Button Menu Constants

NEW WITH THE APPEARANCE MANAGER

You can pass one or more of these constants in the `initialValue` parameter of `NewControl` (page 119) to create a bevel button with a menu of a certain behavior. Bevel buttons with menus have two values: the value of the button and the value of the menu. You can specify the direction of the pop-up menu arrow (down or right) by using the `kControlBevelButtonMenuOnRight` bevel button variant.

```
enum{
    kControlBehaviorCommandMenu     = 0x2000,
    kControlBehaviorMultiValueMenu  = 0x4000
};
```

Constant descriptions

`kControlBehaviorCommandMenu`
: If this bit is set, the menu contains commands, not choices, and should not be marked with a checkmark. If this bit is set, it overrides the `kControlBehaviorMultiValueMenu` bit. This constant is only available with Appearance 1.0.1 and later.

`kControlBehaviorMultiValueMenu`
: If this bit is set, the menus are multi-valued. The bevel button does not maintain the menu value as it normally would (requiring that only one item is selected at a time). This allows the user to toggle entries in a menu and have multiple items checked. In this mode, the menu value accessed with the `kControlMenuLastValueTag` will return the value of the last menu item selected.

Bevel Button and Image Well Content Type Constants

NEW WITH THE APPEARANCE MANAGER

You can use these constants in the `contentType` field of the bevel button and image well content structure (page 107) to display resource or handle-based bevel button and image well content, including text, icon suites, color icons, and pictures.

The resource IDs for icon suite, color icon, and picture resources are passed in the `maximumValue` parameter of `NewControl` (page 119) or in a control `('CNTL')` resource (page 111). The content type is passed in the low byte of the `minimumValue` parameter of `NewControl`.

Note
Resource-based content is owned by the control, while handle-based content is owned by you. The control definition function will not dispose of handle-based content. If you replace handle-based content with resource-based content on the fly, you must dispose of the handle properly to avoid a memory leak.

```
enum {
    kControlContentTextOnly         = 0,
    kControlContentIconSuiteRes     = 1,
    kControlContentCIconRes         = 2,
    kControlContentPictRes          = 3,
    kControlContentIconSuiteHandle  = 129,
    kControlContentCIconHandle      = 130,
    kControlContentPictHandle       = 131,
    kControlContentIconRef          = 132
};
typedef SInt16 ControlContentType;
```

Constant descriptions

`kControlContentTextOnly`
Content type is text only. This constant is passed in the `contentType` field of the bevel button and image well content structure if the content is text only. The variation code `kControlUsesOwningWindowsFontVariant` applies when text content is used.

`kControlContentIconSuiteRes`
: Content type uses an icon suite resource ID. The resource ID of the icon suite resource you wish to display should be in the `resID` field of the bevel button and image well content structure.

`kControlContentCIconRes`
: Content type is a color icon resource ID. The resource ID of the color icon resource you wish to display should be in the `resID` field of the bevel button and image well content structure.

`kControlContentPictRes`
: Content type is a picture resource ID. The resource ID of the picture resource you wish to display should be in the `resID` field of the bevel button and image well content structure.

`kControlContentIconSuiteHandle`
: Content type is an icon suite handle. The handle of the icon suite you wish to display should be in the `iconSuite` field of the bevel button and image well content structure.

`kControlContentCIconHandle`
: Content type uses a color icon handle. The handle of the color icon you wish to display should be in the `cIconHandle` field of the bevel button and image well content structure.

`kControlContentPictHandle`
: Content type uses a picture handle. The handle of the picture you wish to display should be in the `picture` field of the bevel button and image well content structure.

`kControlContentIconRef`
: Reserved. Set to 0.

Bevel Button Graphic Alignment Constants

NEW WITH THE APPEARANCE MANAGER

These constants can be passed in the `inData` parameter of `SetControlData` (page 154) and returned by `GetControlData` (page 156) to specify the placement of icon suites, color icons, and pictures in a bevel button.

```
enum {
    kControlBevelButtonAlignSysDirection    = -1,
    kControlBevelButtonAlignCenter          = 0,
    kControlBevelButtonAlignLeft            = 1,
    kControlBevelButtonAlignRight           = 2,
    kControlBevelButtonAlignTop             = 3,
    kControlBevelButtonAlignBottom          = 4,
    kControlBevelButtonAlignTopLeft         = 5,
    kControlBevelButtonAlignBottomLeft      = 6,
    kControlBevelButtonAlignTopRight        = 7,
    kControlBevelButtonAlignBottomRight     = 8
};
typedef SInt16 ControlButtonGraphicAlignment;
```

Constant descriptions

`kControlBevelButtonAlignSysDirection`
: Bevel button graphic is aligned according to the system default script direction (only left or right).

`kControlBevelButtonAlignCenter`
: Bevel button graphic is aligned center.

`kControlBevelButtonAlignLeft`
: Bevel button graphic is aligned left.

`kControlBevelButtonAlignRight`
: Bevel button graphic is aligned right.

`kControlBevelButtonAlignTop`
: Bevel button graphic is aligned top.

`kControlBevelButtonAlignBottom`
: Bevel button graphic is aligned bottom.

`kControlBevelButtonAlignTopLeft`
: Bevel button graphic is aligned top left.

`kControlBevelButtonAlignBottomLeft`
: Bevel button graphic is aligned bottom left.

`kControlBevelButtonAlignTopRight`
: Bevel button graphic is aligned top right.

`kControlBevelButtonAlignBottomRight`
: Bevel button graphic is aligned bottom right.

Bevel Button Text Alignment Constants

NEW WITH THE APPEARANCE MANAGER

These constants can be passed in the `inData` parameter of `SetControlData` (page 154) and returned by `GetControlData` (page 156) to specify the alignment of text in a bevel button.

```
enum {
    kControlBevelButtonAlignTextSysDirection     = teFlushDefault,
    kControlBevelButtonAlignTextCenter           = teCenter,
    kControlBevelButtonAlignTextFlushRight       = teFlushRight,
    kControlBevelButtonAlignTextFlushLeft        = teFlushLeft
};
typedef SInt16 ControlButtonTextAlignment;
```

Constant descriptions

`kControlBevelButtonAlignTextSysDirection`
Bevel button text is aligned according to the current script direction (left or right).

`kControlBevelButtonAlignTextCenter`
Bevel button text is aligned center.

`kControlBevelButtonAlignTextFlushRight`
Bevel button text is aligned flush right.

`kControlBevelButtonAlignTextFlushLeft`
Bevel button text is aligned flush left.

Bevel Button Text Placement Constants

NEW WITH THE APPEARANCE MANAGER

These constants can be passed in the `inData` parameter of `SetControlData` (page 154) and returned by `GetControlData` (page 156) to specify the placement of bevel button text in relation to an icon suite, color icon, or picture. They can be used in conjunction with bevel button text and graphic alignment constants to create, for example, a button where the graphic and text are left justified with the text below the graphic.

```
enum {
    kControlBevelButtonPlaceSysDirection        = -1,
    kControlBevelButtonPlaceNormally            = 0,
    kControlBevelButtonPlaceToRightOfGraphic    = 1,
    kControlBevelButtonPlaceToLeftOfGraphic     = 2,
    kControlBevelButtonPlaceBelowGraphic        = 3,
    kControlBevelButtonPlaceAboveGraphic        = 4
};
typedef SInt16 ControlButtonTextPlacement;
```

Constant descriptions

`kControlBevelButtonPlaceSysDirection`
: Bevel button text is placed according to the system default script direction.

`kControlBevelButtonPlaceNormally`
: Bevel button text is centered.

`kControlBevelButtonPlaceToRightOfGraphic`
: Bevel button text is placed to the right of the graphic.

`kControlBevelButtonPlaceToLeftOfGraphic`
: Bevel button text is placed to the left of the graphic.

`kControlBevelButtonPlaceBelowGraphic`
: Bevel button text is placed below the graphic.

`kControlBevelButtonPlaceAboveGraphic`
: Bevel button text is placed above the graphic.

Clock Value Flag Constants

NEW WITH THE APPEARANCE MANAGER

You can pass one or more of these mask constants into the control (`'CNTL'`) resource (page 111) or in the `initialValue` parameter of `NewControl` (page 119). The clock control is editable and supports keyboard focus. The little arrows used to allow manipulation of date and time are part of the control, not a separate embedded little arrows control.

```
enum {
    kControlClockNoFlags        = 0,
    kControlClockIsDisplayOnly  = 1,
    kControlClockIsLive         = 2
};
```

Constant descriptions

`kControlClockNoFlags`
Indicates that clock is editable but does not display the current "live" time.

`kControlClockIsDisplayOnly`
When only this bit is set, the clock is not editable. When this bit and the `kControlClockIsLive` bit is set, the clock automatically updates on idle (clock will have the current time).

`kControlClockIsLive`
When only this bit is set, the clock automatically updates on idle and any changes to the clock affect the system clock. When this bit and the `kControlClockIsDisplayOnly` bit is set, the clock automatically updates on idle (clock will have the current time), but is not editable.

Control Part Code Constants

CHANGED WITH THE APPEARANCE MANAGER

Constants of type `ControlPartCode` are returned in the parameter of `FindControl` (page 142) to determine whether a mouse-down event occurred in an active control and, if so, which control.

IMPORTANT

`FindControl` does not usually return the `kControlDisabledPart` or `kControlInactivePart` part codes and never returns them with the standard controls. These are used with `HiliteControl` (page 138).

```
enum {
    kControlNoPart                  = 0,
    kControlLabelPart               = 1,
```

```
    kControlMenuPart                = 2,
    kControlTrianglePart            = 4,
    kControlEditTextPart            = 5,
    kControlPicturePart             = 6,
    kControlIconPart                = 7,
    kControlClockPart               = 8,
    kControlButtonPart              = 10,
    kControlCheckBoxPart            = 11,
    kControlRadioButtonPart         = 12,
    kControlUpButtonPart            = 20,
    kControlDownButtonPart          = 21,
    kControlPageUpPart              = 22,
    kControlPageDownPart            = 23,
    kControlListBoxPart             = 24,
    kControlListBoxDoubleClickPart  = 25,
    kControlImageWellPart           = 26.
    kControlRadioGroupPart          = 27,
    kControlIndicatorPart           = 129,
    kControlDisabledPart            = 254,
    kControlInactivePart            = 255
};
typedef SInt16 ControlPartCode;
```

Constant descriptions

`kControlNoPart` Event did not occur in any control. Also unhighlights any highlighted part of the control when passed to the `HiliteControl` (page 138) function. For bevel buttons with a menu attached, this part code indicates that either the mouse was released outside the bevel button and menu or that the button was disabled.

`kControlLabelPart` Event occurred in the label of a pop-up menu control.

`kControlMenuPart` Event occurred in the menu of a pop-up menu control. For bevel buttons with a menu attached, this part code indicates that the event occurred in a menu item of the bevel button.

`kControlTrianglePart`
Event occurred in a disclosure triangle control.

`kControlEditTextPart`
Event occurred in an editable text control.

`kControlPicturePart`
: Event occurred in a picture control.

`kControlIconPart`
: Event occurred in an icon control.

`kControlClockPart`
: Event occurred in a clock control.

`kControlButtonPart`
: Event occurred in either a push button or bevel button control. For bevel buttons with a menu attached, this part code indicates that the event occurred in the button but not in the attached menu.

`kControlCheckBoxPart`
: Event occurred in a checkbox control.

`kControlRadioButtonPart`
: Event occurred in a radio button control.

`kControlUpButtonPart`
: Event occurred in the up button of a scroll bar control (the arrow at the top or the left).

`kControlDownButtonPart`
: Event occurred in the down button of a scroll bar control (the arrow at the right or the bottom).

`kControlPageUpPart`
: Event occurred in the page-up part of a scroll bar control.

`kControlPageDownPart`
: Event occurred in the page-down part of a scroll bar control.

`kControlListBoxPart`
: Event occurred in a list box control.

`kControlListBoxDoubleClickPart`
: Double-click occurred in a list box control.

`kControlImageWellPart`
: Event occurred in an image well control.

`kControlRadioGroupPart`
: Event occurred in a radio group control. This constant is only available with Appearance 1.0.1 and later.

`kControlIndicatorPart`
: Event occurred in the scroll box of a scroll bar control.

`kControlDisabledPart`
: Used with `HiliteControl` (page 138) to disable the control.

`kControlInactivePart`

Used with `HiliteControl` (page 138) to make the control inactive.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Only the following part codes are supported:

```
kControlNoPart                = 0,
kControlLabelPart             = 1,
kControlMenuPart              = 2,
kControlTrianglePart          = 4,
kControlButtonPart            = 10,
kControlCheckBoxPart          = 11,
kControlRadioButtonPart       = 12,
kControlUpButtonPart          = 20,
kControlDownButtonPart        = 21,
kControlPageUpPart            = 22,
kControlPageDownPart          = 23,
kControlIndicatorPart         = 129,
kControlDisabledPart          = 254,
kControlInactivePart          = 255
```

Part Identifier Constants

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard controls, part identifier constants are ignored and the colors are determined by the current theme.

If you are creating your own control definition function, you can still use these constants in the `partIdentifier` field of a control color table structure to draw a control using colors other than the system default and to identify the part of a control that a color affects.

Meta Font Constants

NEW WITH THE APPEARANCE MANAGER

You can use these constants in the `font` field of the control font style structure (page 103) and the Font ID field of a dialog font table resource (page 254) to specify the style, size, and font family of the control font. You should use these meta font constants whenever possible because the system font can change, depending upon the current theme. If none of these constants are specified, the control uses the system font unless directed to use a window font by a control variant.

```
enum {
    kControlFontBigSystemFont       = -1,
    kControlFontSmallSystemFont     = -2,
    kControlFontSmallBoldSystemFont = -3
};
```

Constant descriptions

`kControlFontBigSystemFont`
Use the system font.

`kControlFontSmallSystemFont`
Use the small system font.

`kControlFontSmallBoldSystemFont`
Use the small emphasized system font (emphasis applied correctly for locale).

The Control Font Style Structure

NEW WITH THE APPEARANCE MANAGER

You can pass a pointer to the control font style structure in the `inStyle` parameter of `SetControlFontStyle` (page 159) to specify a control's font. If none of the flags in the `flags` field of the structure are set, the control uses the system font unless the control variant `kControlUsesOwningWindowsFontVariant` has been specified, in which case the control uses the window font.

If you wish to specify the font for controls in a dialog box, use a dialog font table resource, which is automatically read in by the Dialog Manager; see "The Dialog Control Font Table Resource" (page 254).

A control font style structure is of type `ControlFontStyleRec`:

```
struct ControlFontStyleRec {
SInt16                      flags;
SInt16                      font;
SInt16                      size;
SInt16                      style;
SInt16                      mode;
SInt16                      just;
RGBColor                foreColor;
RGBColor                backColor;
};
typedef struct ControlFontStyleRec ControlFontStyleRec;
typedef ControlFontStyleRec *ControlFontStylePtr;
```

Field descriptions

`flags` A signed 16-bit integer specifying which fields of the structure should be applied to the control; see "Control Font Style Flag Constants" (page 105). If none of the flags in the `flags` field of the structure are set, the control uses the system font unless the control variant `kControlUsesOwningWindowsFontVariant` has been specified, in which case the control uses the window font.

`font` If the `kControlUseFontMask` bit is set, then this element will contain an integer indicating the ID of the font family to use. If this bit is not set, then the system default font is used. A meta font constant can be specified instead; see "Meta Font Constants" (page 103).

`size` If the `kControlUseSizeMask` bit is set, then this element will contain an integer representing the point size of the text. If the `kControlAddSizeMask` bit is set, this value will represent the size to add to the current point size of the text. A meta font constant can be specified instead; see "Meta Font Constants" (page 103).

`style` If the `kControlUseStyleMask` bit is set, then this element will contain an integer specifying which styles to apply to the

text. If all bits are clear, 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

`mode` If the `kControlUseModeMask` bit is set, then this element will contain an integer specifying how characters are drawn in the bit image. See *Inside Macintosh: Imaging With QuickDraw* for a discussion of transfer modes.

`just` If the `kControlUseJustMask` bit is set, then this element will contain an integer specifying text justification (left, right, centered, or system script direction).

`foreColor` If the `kControlUseForeColorMask` bit is set, then this element will contain an RGB color to use when drawing the text.

`backColor` If the `kControlUseBackColorMask` bit is set, then this element will contain an RGB color to use when drawing the background behind the text. In certain text modes, background color is ignored.

Control Font Style Flag Constants

NEW WITH THE APPEARANCE MANAGER

You can pass one or more of these flag constants in the `flags` field of the control font style structure (page 103) to specify the field(s) of the structure that should be applied to the control. If none of the flags are set, the control uses the system font unless a control variant specifies use of a window font.

```
enum {
    kControlUseFontMask         = 0x0001,
    kControlUseFaceMask         = 0x0002,
    kControlUseSizeMask         = 0x0004,
    kControlUseForeColorMask    = 0x0008,
    kControlUseBackColorMask    = 0x0010,
```

```
    kControlUseModeMask         = 0x0020,
    kControlUseJustMask         = 0x0040,
    kControlUseAllMask          = 0x00FF,
    kControlAddFontSizeMask     = 0x0100
};
```

Constant descriptions

`kControlUseFontMask`
: If the `kControlUseFontMask` flag is set (bit 0), the `font` field of the control font style structure is applied to the control.

`kControlUseFaceMask`
: If the `kControlUseFaceMask` flag is set (bit 1), the `style` field of the control font style structure is applied to the control. This flag is ignored if you specify a meta font value; see "Meta Font Constants" (page 103).

`kControlUseSizeMask`
: If the `kControlUseSizeMask` flag is set (bit 2), the `size` field of the control font style structure is applied to the control. This flag is ignored if you specify a meta font value; see "Meta Font Constants" (page 103).

`kControlUseForeColorMask`
: If the `kControlUseForeColorMask` flag is set (bit 3), the `foreColor` field of the control font style structure is applied to the control. This flag only applies to static text controls.

`kControlUseBackColorMask`
: If the `kControlUseBackColorMask` flag is set (bit 4), the `backColor` field of the control font style structure is applied to the control. This flag only applies to static text controls.

`kControlUseModeMask`
: If the `kControlUseModeMask` flag is set (bit 5), the text mode specified in the `mode` field of the control font style structure is applied to the control.

`kControlUseJustMask`
: If the `kControlUseJustMask` flag is set (bit 6), the `just` field of the control font style structure is applied to the control.

`kControlUseAllMask`
: If `kControlUseAllMask` is used, all flags in this mask will be set except `kControlUseAddFontSizeMask`.

`kControlUseAddFontSizeMask`
If the `kControlUseAddFontSizeMask` flag is set (bit 8), the Dialog Manager will add a specified font size to the `size` field of the control font style structure. This flag is ignored if you specify a meta font value; see "Meta Font Constants" (page 103).

The Bevel Button and Image Well Content Structure

NEW WITH THE APPEARANCE MANAGER

You can pass a pointer to the bevel button and image well content structure of type `ControlButtonContentInfo` in the `inBuffer` parameter of `GetControlData` (page 156) to get the resource ID (for resource-based content) or handle (for handle-based content) of a color icon, picture, or icon suite in a bevel button or image well.

```
struct ControlButtonContentInfo {
    ControlContentType contentType;
    union {
    SInt16              resID;
    CIconHandle         cIconHandle;
    Handle              iconSuite;
    Handle              iconRef;
    PicHandle           picture;
    }                   u;
};
typedef struct ControlButtonContentInfo ControlButtonContentInfo;
typedef ControlButtonContentInfo *ControlButtonContentInfoPtr;
```

Field descriptions

`contentType` Specifies the bevel button or image well content type and whether the content is text-only, resource-based, or handle-based; see "Bevel Button and Image Well Content Type Constants" (page 94). The value specified in the `contentType` field determines which of the other fields in the structure are used.

`resID` If the content type specified in the `contentType` field is `kControlContentIconSuiteRes`, `kControlContentCIconRes`, or

	`kControlContentPictRes`, this field contains the resource ID of a picture, color icon, or icon suite resource.
`cIconHandle`	If the content type specified in the `contentType` field is `kControlContentCIconHandle`, this field contains a handle to a color icon.
`iconSuite`	If the content type specified in the `contentType` field is `kControlContentIconSuiteHandle`, this field contains a handle to an icon suite.
`iconRef`	Reserved.
`picture`	If the content type specified in the `contentType` field is `kControlContentPictHandle`, this field contains a handle to a picture.

The Editable Text Selection Structure

NEW WITH THE APPEARANCE MANAGER

You can pass a pointer to the editable text selection structure to `GetControlData` (page 156) and `SetControlData` (page 154) to access and set the current selection range in an editable text control.

An editable text selection structure is of type `ControlEditTextSelectionRec`:

```
struct ControlEditTextSelectionRec {
    SInt16                      selStart;
    SInt16                      selEnd;
};
typedef struct ControlEditTextSelectionRec ControlEditTextSelectionRec;
typedef ControlEditTextSelectionRec *ControlEditTextSelectionPtr;
```

Field descriptions

`selStart`	A signed 16-bit integer indicating the beginning of the editable text selection.
`selEnd`	A signed 16-bit integer indicating the end of the editable text selection.

The Tab Information Structure

NEW WITH THE APPEARANCE MANAGER

If you are not creating a tab control with a `'tab#'` resource, you can call `SetControlMaximum` to set the number of tabs in a tab control. Then use `SetControlData` (page 154) with the tab information structure to access and set information for an individual tab in a tab control. Available with Appearance 1.0.1 and later.

A tab information structure is of type `ControlTabInfoRec`:

```
struct ControlTabInfoRec {
    SInt16                      version;
    SInt16                      iconSuiteID;
    Str255                      name;
};
```

Field descriptions

`version` A signed 16-bit integer indicating the version of the tab information structure. The only currently available version value is 0.

`iconSuiteID` A signed 16-bit integer indicating the ID of an icon suite to be used for the tab label. Pass 0 for no icon.

`name` A string specifying the title to be used for the tab label.

The Auxiliary Control Structure

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard controls, most of the fields of the auxiliary control structure are ignored except the `acCTable` and `acFlags` fields. If you are creating your own control definition function, the entire auxiliary control structure can be used.

The Pop-Up Menu Private Structure

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, you should not access the pop-up menu private data structure. Instead, you should pass the value `kControlBevelButtonMenuHandleTag` in the `tagName` parameter of `GetControlData` (page 156) to get the menu handle of a bevel button, and the menu handle and the menu ID of the menu associated with a pop-up menu.

The Control Color Table Structure

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard controls, the control color table structure is ignored and the colors are determined by the current theme. If you are creating your own control definition function, you can use the control color table structure to draw a control using colors other than the system default.

Result Codes

The most common result codes returned by Control Manager functions are listed below.

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough memory
`resNotFound`	–192	Unable to read resource
`hmHelpManagerNotInited`	–855	Help menu not set up
`errMessageNotSupported`	–30580	Message not supported
`errDataNotSupported`	–30581	Data not supported
`errControlDoesntSupportFocus`	–30582	Control does not support focus
`errWindowDoesntSupportFocus`	–30583	Window does not support focus
`errUnknownControl`	–30584	Unknown control
`errCouldntSetFocus`	–30585	Could not set focus
`errNoRootControl`	–30586	No root control established
`errRootAlreadyExists`	–30587	Root control already exists
`errInvalidPartCode`	–30588	Invalid part code
`errControlsAlreadyExist`	–30589	Control already exists
`errControlIsNotEmbedder`	–30590	Control is not an embedder
`errDataSizeMismatch`	–30591	Data size mismatch

errControlHiddenOrDisabled	–30592	Control hidden or disabled
errWindowRegionCodeInvalid	–30593	Window region code invalid
errCantEmbedIntoSelf	–30594	Can't embed control in self
errCantEmbedRoot	–30595	Can't embed root control
errItemNotControl	–30596	Dialog item not a control

Control Manager Resources

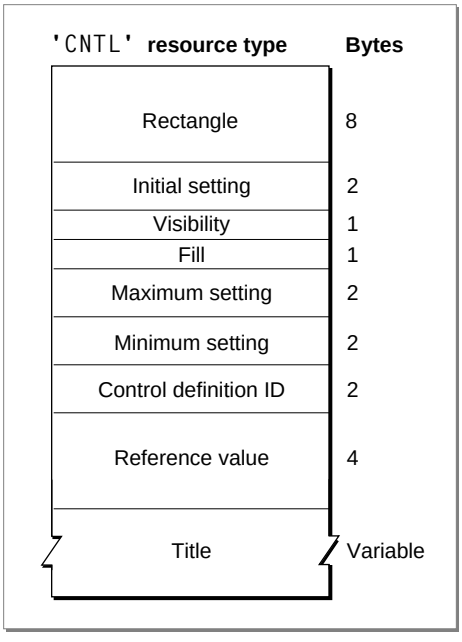
This section describes resources used by the Control Manager for defining and displaying controls. The structures of these resources are described after they are compiled by the Rez resource compiler. To create these resources, you can either specify the resource description in an input file and compile the resource using a resource compiler, such as Rez, or you can directly create your resources in a resource file using a tool such as ResEdit. For examples of Rez input files for these resources, see "Using the Control Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

The Control Resource

CHANGED WITH THE APPEARANCE MANAGER

You can use a control (`'CNTL'`) resource to define a standard control; many new standard controls have been added with the Appearance Manager. All control resources must have resource ID numbers greater than 127. Use `GetNewControl` (page 118) to create a control defined in a control resource. The Control Manager uses the information you specify to create a control structure in memory. Figure 2-1 shows the structure of this resource.

Figure 2-1 Structure of a compiled control ('CNTL') resource



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

- The rectangle, specified in coordinates local to the window, that encloses the control and thus determines its size and location.
- The initial setting for the control; see "Settings Values for Standard Controls" (page 78).
- The visibility of the control. If this element contains the value `true`, `GetNewControl` draws the control immediately, without using the application's standard updating mechanism for windows. If this element contains the value `false`, the application must use `ShowControl` (page 134) when it's prepared to display the control.
- Fill. Set to 0.
- The maximum setting for the control; see "Settings Values for Standard Controls" (page 78).

- The minimum setting for the control; see "Settings Values for Standard Controls" (page 78).
- The control definition ID, which the Control Manager uses to determine the control definition function for this control; see "Control Definition IDs" (page 71).
- The control's reference value, which is set and used only by the application—except when the application adds the `kControlPopupUseAddResMenuVariant` variation code to the `kControlPopupButtonProc` control definition ID.
- For controls that need a title, the string for that title; for controls that don't use titles, an empty string.

Note
The titles of all Appearance-compliant standard system controls appear in the system font. You should generally use the system font or small system font in your controls; see *Mac OS 8 Human Interface Guidelines* for more details.

The Control Definition Function Resource

CHANGED WITH THE APPEARANCE MANAGER

In addition to the standard controls, the Control Manager allows you to define new, nonstandard controls as appropriate for your application. To define your own type of control, write a control definition function which is stored in a resource of type `'CDEF'`. Provide as the resource data the compiled or assembled code of your control definition procedure. The entry point of your procedure must be at the beginning of the resource data. See "Defining Your Own Control Definition Function" (page 162) for more information about creating a control definition function.

The Control Color Table Resource

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard controls, the control color table (`'cctb'`) resource is ignored and the colors are determined by the current theme.

If you are creating your own control definition function, you can still use the control color table structure to draw a control using colors other than the system default.

The List Box Description Resource

NEW WITH THE APPEARANCE MANAGER

You can use a list box description resource to specify information in a list box. A list box description resource is a resource of type `'ldes'`. All list box description resources must have resource ID numbers greater than 127. The Control Manager uses the information you specify to provide additional information to the corresponding list box control. Figure 2-2 shows the structure of this resource.

Figure 2-2 Structure of a compiled list box description (`'ldes'`) resource

`'ldes'` resource type	Bytes
Version number	2
Number of rows	2
Number of columns	2
Cell height	2
Cell width	2
Has vertical scroll	1
Reserved	1
Has horizontal scroll	1
Reserved	1
List definition resource ID	2
Has size box	1
Reserved	1

You define a list box description resource by specifying these elements:

- Version number. An integer specifying the version of the resource format.

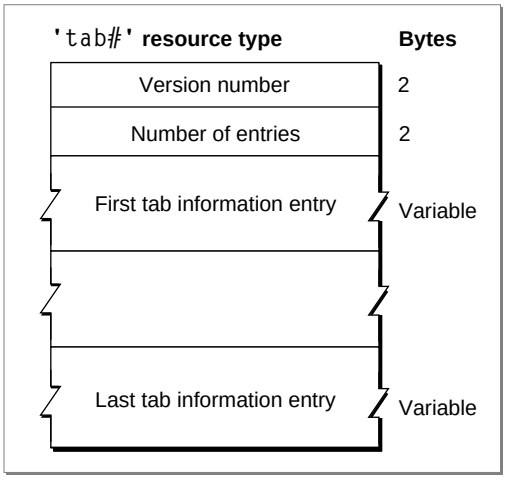
- Number of rows. An integer specifying the number of rows in the list box.
- Number of columns. An integer specifying the number of columns in the list box.
- Cell height. An integer specifying the height of a list item. If 0 is specified, the list item height is automatically calculated.
- Cell width. An integer specifying the width of a list item. If 0 is specified, the list item width is automatically calculated.
- Has vertical scroll bar. A Boolean value that indicates whether the list box should contain a vertical scroll bar. If `true`, the list box contains a vertical scroll bar; if `false`, no vertical scroll bar.
- Reserved. Set to 0.
- Has horizontal scroll bar. A Boolean value that indicates whether the list should contain a horizontal scroll bar. Specify `true` if your list requires a horizontal scroll bar; specify `false` otherwise.
- Reserved. Set to 0.
- Resource ID. This is the resource ID of the list definition procedure to use for the list. To use the default list definition procedure, which supports the display of unstyled text, specify a resource ID of 0.
- Has size box. A Boolean value that indicates whether the List Manager should leave room for a size box. If `true`, a size box will be drawn; if `false`, a size box will not be drawn.
- Reserved. Set to 0.

The Tab Information Resource

NEW WITH THE APPEARANCE MANAGER

You can use a tab information resource to specify the icon suite ID and name of each tab in a tab control. A tab information resource is a resource of type `'tab#'`. All tab information resources must have resource ID numbers greater than 127. The Control Manager uses the information you specify to provide additional information to the corresponding tab control. Figure 2-3 shows the structure of this resource.

Figure 2-3 Structure of a compiled tab information (`'tab#'`) resource

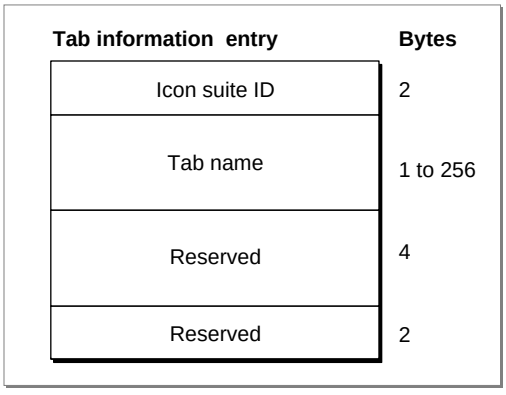


A compiled version of a tab information resource contains the following elements:

- Version number. An integer specifying the version of the resource.
- An integer that specifies the number of entries in the resource (that is, the number of tab information structures).
- A series of tab information structures, each of which consists of a 2-byte icon suite identifier and a variable-length string indicating the tab name.

Figure 2-4 shows the format of a compiled entry in a `'tab#'` resource. A tab information entry specifies the icon suite ID and the name of a tab control.

Figure 2-4 Structure of a tab information entry



Each entry in a `'tab#'` resource contains the following:

- Icon suite ID. A value of 0 indicates no icon.
- Tab name. The title of the tab control.
- Reserved. Set to 0.
- Reserved. Set to 0.

Control Manager Functions

Creating and Removing Controls

GetNewControl

CHANGED WITH THE APPEARANCE MANAGER

Creates a control from a description in a control (`'CNTL'`) resource.

```
pascal ControlHandle GetNewControl (
                    SInt16 resourceID,
                    WindowPtr owningWindow);
```

`resourceID` The resource ID of a control resource; see Table 2-1 (page 72).

`owningWindow` A pointer to the window in which you want to place the control.

function result Returns a handle to the control created from the specified control resource. If `GetNewControl` can't read the control resource from the resource file, it returns `nil`.

DISCUSSION

The `GetNewControl` function creates a control structure from the information in the specified control resource, adds the control structure to the control list for the specified window, and returns as its function result a handle to the control. You use this handle when referring to the control in most other Control Manager functions. After making a copy of the control resource, `GetNewControl` releases the memory occupied by the original control resource before returning.

The control resource specifies the rectangle for the control, its initial setting, its visibility state, its maximum and minimum settings, its control definition ID, a reference value, and its title (if any). After you use `GetNewControl` to create the

control, you can change the control characteristics with other Control Manager functions.

If the control resource specifies that the control should be visible, the Control Manager draws the control. If the control resource specifies that the control should initially be invisible, you can use the function `ShowControl` (page 134) to make the control visible.

When an embedding hierarchy is established within a window, `GetNewControl` automatically embeds the newly created control in the root control of the owning window. See "Embedding Controls" (page 123).

If you are using standard system controls, default colors are used and the control color table resource is ignored. To use colors other than the default colors, you must write your own custom control definition function.

SEE ALSO

`NewControl` (page 119).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`GetNewControl` does not embed the newly created control in the root control of the owning window because embedding hierarchies are not supported.

NewControl

CHANGED WITH THE APPEARANCE MANAGER

Creates a control based on information passed in its parameters that describes the control.

```
pascal ControlHandle NewControl (
                    WindowPtr owningWindow,
                    const Rect *boundsRect,
                    ConstStr255Param controlTitle,
                    Boolean initiallyVisible,
                    SInt16 initialValue,
                    SInt16 minimumValue,
```

```
                    SInt16 maximumValue,
                    SInt16 procID,
                    SInt32 controlReference);
```

`owningWindow` On input, a pointer to the window in which you want to place the control. All coordinates pertaining to the control are interpreted in this window's local coordinate system.

`boundsRect` On input, a pointer to a rectangle, specified in the given window's local coordinates, that encloses the control and thus determines its size and location. When specifying this rectangle, you should follow the guidelines presented in "Dialog Box Layout", in *Mac OS 8 Human Interface Guidelines,* for control placement and alignment.

`controlTitle` The title string, used for push buttons, checkboxes, radio buttons, and pop-up menus. When specifying a multiple-line title, separate the lines with the ASCII character code 0x0D (carriage return). For controls that don't use titles, pass an empty string.

`initiallyVisible`
A Boolean value specifying the visible/invisible state for the control. If you pass `true` in this parameter, `NewControl` draws the control immediately, without using your window's standard updating mechanism. If you pass `false`, you must later use `ShowControl` (page 134) to display the control.

`initialValue` The initial setting for the control; see "Settings Values for Standard Controls" (page 78).

`minimumValue` The minimum setting for the control; see "Settings Values for Standard Controls" (page 78).

`maximumValue` The maximum setting for the control; see "Settings Values for Standard Controls" (page 78).

`procID` The control definition ID; see Table 2-1 (page 72). If the control definition function isn't in memory, it is read in.

`controlReference`
The control's reference value, which is set and used only by your application.

function result Returns a handle to the control described in its parameters. If `NewControl` runs out of memory or fails, it returns `nil`.

DISCUSSION

The `NewControl` function creates a control structure from the information you specify in its parameters, adds the control structure to the control list for the specified window, and returns as its function result a handle to the control. You can use this handle when referring to the control in most other Control Manager functions. Generally, you should use the function `GetNewControl` (page 118) instead of `NewControl`, because `GetNewControl` is a resource-based control-creation function that allows you to localize your application without recompiling.

When an embedding hierarchy is established within a window, `NewControl` automatically embeds the newly created control in the root control of the owning window. See "Embedding Controls" (page 123).

If you are using standard system controls, default colors are used and the control color table resource is ignored. To use colors other than the default colors, write your own custom control definition function.

SEE ALSO

`GetNewControl` (page 118).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`NewControl` does not embed the newly created control in the root control of the owning window because embedding hierarchies are not supported.

DisposeControl

CHANGED WITH THE APPEARANCE MANAGER

Removes a particular control and its embedded controls from a window that you wish to keep.

```
pascal void DisposeControl (ControlHandle theControl);
```

`theControl` On input, a handle to the control you wish to remove.

DISCUSSION

The `DisposeControl` function removes the specified control (and any embedded controls it may possess) from the screen, deletes it from the window's control list, and releases the memory occupied by the control structure and any data structures associated with the control. Passing the root control to this function is the effectively the same as calling `KillControls` (page 122). If an embedding hierarchy is present, `DisposeControl` disposes of the controls embedded within a control before disposing of the container control.

SPECIAL CONSIDERATIONS

The Window Manager functions `CloseWindow` and `DisposeWindow` automatically dispose of all controls associated with the given window.

SEE ALSO

"Embedding Controls" (page 123).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`DisposeControl` does not dispose of embedded controls, because embedding hierarchies are not supported.

KillControls

CHANGED WITH THE APPEARANCE MANAGER

Removes all controls in a specified window.

```
pascal void KillControls (WindowPtr theWindow);
```

`theWindow` On input, a pointer to the window whose controls you wish to remove.

DISCUSSION

The `KillControls` function disposes of all controls associated with the specified window. To remove just one control, use `DisposeControl` (page 121). If an

embedding hierarchy is present, `KillControls` disposes of the controls embedded within a control before disposing of the container control.

SPECIAL CONSIDERATIONS

The Window Manager functions `CloseWindow` and `DisposeWindow` automatically dispose of all controls associated with the given window.

SEE ALSO

"Embedding Controls" (page 123).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`KillControls` does not dispose of embedded controls, because embedding hierarchies are not supported.

Embedding Controls

This section provides functions that you can use to establish an embedding hierarchy. This can be accomplished in two steps: creating a root control and embedding controls within it.

To embed controls in a window, you must create a root control for that window. The **root control** is the container for all other window controls. You create the root control in one of two ways—by calling the `CreateRootControl` (page 125) function or by setting the appropriate dialog flag. The root control can be retrieved by calling `GetRootControl` (page 126).

The root control is implemented as a user pane control. You can attach any application-defined user pane functions to the root control to perform actions such as hit testing, drawing, handling keyboard focus, erasing to the correct background, and processing idle and keyboard events. For information on how to write these functions, see "Defining Your Own User Pane Functions" (page 189).

Once you have created a root control, newly created controls will automatically be embedded in the root control when you call `NewControl` (page 119) or `GetNewControl` (page 118). You can specify that a specific control be embedded into another by calling `EmbedControl` (page 127).

By acting on an embedder control, you can move, disable, or hide groups of items. For example, you can use a blank user pane control as the embedder control for all items in a particular "page" of a tab control. After creating as many user panes as you have tabs, you can hide one and show the next when a tab is clicked. All the controls embedded in the user pane will be hidden and shown automatically when the user pane is hidden and shown.

The Dialog Manager uses `AutoEmbedControl` (page 128) to position dialog items in an **embedding hierarchy** based on both visual containment and the item list resource order. As items are added to a dialog box during creation, controls that already exist in the window will be containers for new controls if they both visually contain the control and have set the `kControlSupportsEmbedding` feature bit. For this reason, you should place the largest embedder controls at the beginning of the item list resource. As an example, the Dialog Manager would embed radio buttons in a tab control if they visually "fit" inside the tab control, as long as the tab control was already created in a `'DITL'` resource and established as an embedder control.

In addition to calling `CreateRootControl`, you can establish an embedding hierarchy in a dialog box by either setting the feature bit `kDialogFlagsUseControlHierarchy` in the extended dialog resource (page 252) or passing it in the `inFlags` parameter of `NewFeaturesDialog` (page 270). An embedding hierarchy can be created in an alert box by setting the `kAlertFlagsUseControlHierarchy` bit in the extended alert resource (page 253). It is important to note that a preexisting alert or dialog item will become a control if it is in an alert or dialog box that now uses an embedding hierarchy.

The embedding hierarchy enforces drawing order by drawing the embedding control before its embedded controls. Using an embedding hierarchy also enforces orderly hit-testing, since it performs an "inside-out" hit test to determine the most deeply nested control that is hit by the mouse. An embedding hierarchy is also necessary for controls to make use of keyboard focus, the default focusing order for which is a linear progression that uses the order the controls were added to the window. For more details on keyboard focus, see "Handling Keyboard Focus" (page 147).

CreateRootControl

NEW WITH THE APPEARANCE MANAGER

Creates the root control for a specified window.

```
pascal OSErr CreateRootControl (
                    WindowPtr inWindow,
                    ControlHandle* outControl);
```

`inWindow`	On input, a pointer to the window in which you wish to create a root control.
`outControl`	On output, a pointer to a handle to the root control.
function result	A result code; see "Result Codes" (page 110). The result code `errControlsAlreadyExist` indicates that other controls were already present when `CreateRootControl` was called. The result code `errRootAlreadyExists` indicates that a root control was already created for the window.

DISCUSSION

The `CreateRootControl` function creates the root control for a window if no other controls are present. If there are any controls in the window prior to calling `CreateRootControl`, an error is returned and the root control is not created.

The root control acts as the top-level container for a window and is required for embedding to occur. Once the root control is created, you can call `EmbedControl` (page 127) and `AutoEmbedControl` (page 128) to embed controls in the root control.

Note
The minimum, maximum, and initial settings for a root control are reserved and should not be changed.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

GetRootControl

NEW WITH THE APPEARANCE MANAGER

Returns a handle to a window's root control.

```
pascal OSErr GetRootControl (
                    WindowPtr inWindow,
                    ControlHandle* outControl);
```

`inWindow` On input, a pointer to the root control's owning window.

`outControl` On output, a pointer to a handle to the root control.

function result A result code; see "Result Codes" (page 110). The result code `errNoRootControl` indicates that there is no root control in the specified window.

DISCUSSION

You can call `GetRootControl` to determine whether or not a root control (and therefore an embedding hierarchy) exists within a specified window. Once you have the root control's handle, you can pass it to functions such as `DisposeControl` (page 121), `ActivateControl` (page 135), and `DeactivateControl` (page 136) to apply specified actions to the entire embedding hierarchy.

Note
The minimum, maximum, and initial settings for a root control are reserved and should not be changed.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

EmbedControl

NEW WITH THE APPEARANCE MANAGER

Embeds one control inside another.

```
pascal OSErr EmbedControl (
                    ControlHandle inControl,
                    ControlHandle inContainer);
```

`inControl` On input, a handle to a control to be embedded.

`inContainer` On input, a handle to the embedder control.

function result A result code; see "Result Codes" (page 110). The result code `errNoRootControl` indicates that there is no root control in the specified window. The result code `errControlIsNotEmbedder` indicates that the specified control does not support embedding. The result code `errCantEmbedIntoSelf` indicates that the controls specified in the `inControl` and `inContainer` parameters are the same control. The result code `errCantEmbedRoot` indicates that you are trying to embed the root control.

DISCUSSION

An embedding hierarchy must be established before your application calls the `EmbedControl` function. If the specified control does not support embedding or there is no root control in the owning window, an error is returned. If the control you wish to embed is in a different window from the embedder control, an error is returned. See "Embedding Controls" (page 123) for more details on embedding.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`AutoEmbedControl` (page 128).

AutoEmbedControl

NEW WITH THE APPEARANCE MANAGER

Automatically embeds a control in the smallest appropriate embedder control.

```
pascal OSErr AutoEmbedControl (
                    ControlHandle inControl,
                    WindowPtr inWindow);
```

`inControl` On input, a handle to a control to be embedded.

`inWindow` On input, a pointer to the window in which you want to embed the control.

function result A result code; see "Result Codes" (page 110). The result code `errControlIsNotEmbedder` indicates that embedding is not enabled for that window. The result code `errItemNotControl` indicates that the dialog item that you wish to embed is not a control (not in an embedding hierarchy).

DISCUSSION

The Dialog Manager uses `AutoEmbedControl` (page 128) to position dialog items in an embedding hierarchy based on both visual containment and the item list resource order. For information on embedding hierarchies in dialog and alert boxes, see "Embedding Controls" (page 123).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`EmbedControl` (page 127).

CountSubControls

NEW WITH THE APPEARANCE MANAGER

Returns the number of embedded controls within a control.

```
pascal OSErr CountSubControls (
                    ControlHandle inControl,
                    SInt16* outNumChildren);
```

`inControl` On input, a handle to a control whose embedded controls you wish to count.

`outNumChildren` On output, a pointer to an integer representing the number of embedded subcontrols.

function result A result code; see "Result Codes" (page 110). The result code `errControlIsNotEmbedder` indicates that the specified control does not support embedding. The result `errNoRootControl` indicates that embedding is not enabled for that window.

DISCUSSION

The `CountSubControls` function is useful for iterating over the control hierarchy. You can use the count produced to determine how many subcontrols there are and then call `GetIndexedSubControl` (page 130) to get each.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

GetIndexedSubControl

NEW WITH THE APPEARANCE MANAGER

Returns a handle to a specified embedded control.

```
pascal OSErr GetIndexedSubControl (
                    ControlHandle inControl,
                    SInt16 inIndex,
                    ControlHandle* outSubControl);
```

`inControl` On input, a handle to an embedder control.

`inIndex` On input, a 1-based index—an integer between 1 and the value returned in the `outNumChildren` parameter of `CountSubControls` (page 129)—indicating the specific control you wish to access.

`outSubControl` On output, a pointer to a handle to the embedded control.

function result A result code; see "Result Codes" (page 110). The result code `errControlIsNotEmbedder` indicates that the embedder control does not support embedding or that embedding is not enabled for that window. If the index passed in is invalid, the `paramErr` result code is returned.

DISCUSSION

The `GetIndexedSubControl` function is useful for iterating over the control hierarchy. Also, the value of a radio group control is the index of its currently selected embedded radio button control. So, passing the current value of a radio group control into `GetIndexedSubControl` will give you a handle to the currently selected radio button control.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

GetSuperControl

NEW WITH THE APPEARANCE MANAGER

Produces a handle to the embedder control.

```
pascal OSErr GetSuperControl (
                    ControlHandle inControl,
                    ControlHandle* outParent);
```

`inControl` On input, a handle to an embedded control.

`outParent` On output, a pointer to a handle to the embedder control.

function result A result code; see "Result Codes" (page 110). The result code `errControlIsNotEmbedder` indicates that the specified control does not support embedding. The result code `errCantEmbedRoot` indicates that you passed the root control in the `inControl` parameter.

DISCUSSION

The `GetSuperControl` function gets a handle to the parent control of the control passed in.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

SetControlSupervisor

NEW WITH THE APPEARANCE MANAGER

Routes mouse-down events to the embedder control.

```
pascal OSErr SetControlSupervisor (
                    ControlHandle inControl,
                    ControlHandle inBoss);
```

`inControl` On input, a handle to an embedded control.

`inBoss`	On input, a handle to the embedder control that you wish to route mouse-down events to.
function result	A result code. The result code `errControlIsNotEmbedder` indicates that the control specified in the `inBoss` parameter is not an embedder control. For a list of other result codes, see "Result Codes" (page 110).

DISCUSSION

The `SetControlSupervisor` function allows an embedder control to respond to mouse-down events occurring in its embedded controls.

An example of a standard control that uses this function is the radio group control. Mouse-down events in the embedded controls of a radio group are intercepted by the group control. (The embedded controls in this case must support radio behavior; if a mouse-down event occurs in an embedded control within a radio group control that does not support radio behavior, the control tracks normally and the group is not involved.) The group handles all interactions and switches the embedded control's value on and off. If the value of the radio group changes, `TrackControl` (page 146) or `HandleControlClick` (page 144) will return the `kControlRadioGroupPart` part code. If the user tracks off the radio button or clicks the current radio button, `kControlNoPart` is returned.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

DumpControlHierarchy

NEW WITH THE APPEARANCE MANAGER

Writes a textual representation of the control hierarchy for a specified window into a file.

```
pascal OSErr DumpControlHierarchy (
                    WindowPtr inWindow,
                    const FSSpec* inDumpFile);
```

`inWindow`	On input, a pointer to the window whose control hierarchy you wish to debug.
`inDumpFile`	On input, a pointer to a file specification in which you wish to place a text description of the window's control hierarchy.
function result	A result code; see "Result Codes" (page 110).

DISCUSSION

The `DumpControlHierarchy` function places a text listing of the current control hierarchy for the window specified into the specified file, overwriting any existing file. If the specified window does not contain a control hierarchy, `DumpControlHierarchy` notes this in the text file. This function is useful for debugging embedding-related problems.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

Manipulating Controls

When showing, hiding, activating, or deactivating groups of controls, the state of an embedded control that is hidden or deactivated is preserved so that when the embedder control is shown or activated, the embedded control appears in the same state as the embedder. An embedded control is considered **latent** when it is deactivated or hidden due to its embedder control being deactivated or hidden. If you activate a latent embedded control whose embedder is deactivated, the embedded control becomes latent until the embedder is activated. However, if you deactivate a latent embedded control, it will not be activated when its embedder is activated.

When activating and deactivating controls in an embedding hierarchy, call `ActivateControl` (page 135) and `DeactivateControl` (page 136) instead of `HiliteControl` (page 138) to ensure that latent embedded controls are displayed correctly.

ShowControl

CHANGED WITH THE APPEARANCE MANAGER

Makes an invisible control, and any latent embedded controls, visible.

```
pascal void ShowControl (ControlHandle theControl);
```

`theControl` On input, a handle to the control you want to make visible.

DISCUSSION

If the specified control is invisible, the `ShowControl` function makes it visible and immediately draws the control within its window without using your window's standard updating mechanism. If the specified control has embedded controls, `ShowControl` makes the embedded controls visible as well. If the control is already visible, `ShowControl` has no effect.

If you call `ShowControl` on a latent embedded control whose embedder is disabled, the embedded control will be invisible until its embedder control is enabled. For a discussion of latency, see "Manipulating Controls" (page 133).

You can make a control invisible in several ways:

- Specifying its invisibility in the control resource.
- Passing a value of `false` in the `visible` parameter of `NewControl` (page 119).
- Calling `HideControl` (page 135).
- Calling `SetControlVisibility` (page 160). The setting takes effect the next time the control is drawn.

SPECIAL CONSIDERATIONS

The `ShowControl` function draws the control in its window, but the control can still be completely or partially obscured by overlapping windows or other objects.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`ShowControl` only makes the specified control visible, because embedding hierarchies are not supported.

HideControl

CHANGED WITH THE APPEARANCE MANAGER

Makes the visible control, and any latent embedded controls, invisible.

```
pascal void HideControl (ControlHandle theControl);
```

`theControl` On input, a handle to the control you want to hide.

DISCUSSION

The `HideControl` function makes the specified control invisible. This can be useful, for example, before adjusting a control's size and location. It also adds the control's rectangle to the window's update region, so that anything else that was previously obscured by the control will reappear on the screen. If the specified control has embedded controls, `HideControl` makes the embedded controls invisible as well. If the control is already invisible, `HideControl` has no effect.

If you call `HideControl` on a latent embedded control, it would not be displayed the next time `ShowControl` was called on its embedder control. For a discussion of latency, see "Manipulating Controls" (page 133).

To make the control visible again, you can use the `ShowControl` function (page 134) or `SetControlVisibility` (page 160).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`HideControl` only makes the specified control invisible, because embedding hierarchies are not supported.

ActivateControl

NEW WITH THE APPEARANCE MANAGER

Activates a control and any latent embedded controls.

```
pascal OSErr ActivateControl (ControlHandle inControl);
```

`inControl`	On input, a handle to a control that you wish to activate. Passing a window's root control will activate all controls in that window.
function result	A result code; see "Result Codes" (page 110).

DISCUSSION

The `ActivateControl` function should be called instead of `HiliteControl` to activate a specified control and its latent embedded controls. For a discussion of latency, see "Manipulating Controls" (page 133).

You can use `ActivateControl` to activate all controls in a window by passing the window's root control in the `inControl` parameter.

If a control definition function supports activate events, it will receive a `kControlMsgActivate` message before redrawing itself in its active state.

SEE ALSO

`DeactivateControl` (page 136).

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

DeactivateControl

NEW WITH THE APPEARANCE MANAGER

Deactivates a control and any latent embedded controls.

```
pascal OSErr DeactivateControl (ControlHandle inControl);
```

`inControl`	On input, a handle to the control that you wish to deactivate. Passing a window's root control will deactivate all controls in that window.
function result	A result code; see "Result Codes" (page 110).

DISCUSSION

The `DeactivateControl` function should be called instead of `HiliteControl` to deactivate a specified control and its latent embedded controls. For a discussion of latency, see "Manipulating Controls" (page 133).

You can use `DeactivateControl` to deactivate all controls in a window by passing the window's root control in the `inControl` parameter.

If a control definition function supports activate events, it will receive a `kControlMsgActivate` message before redrawing itself in its inactive state.

SEE ALSO

`ActivateControl` (page 135).

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

IsControlActive

NEW WITH THE APPEARANCE MANAGER

Indicates whether a control is active.

```
pascal Boolean IsControlActive (ControlHandle inControl);
```

`inControl`	On input, a handle to the control whose activity you wish to determine.
function result	Returns a Boolean value. If `true`, the control is active. If `false`, the control is inactive.

DISCUSSION

If you wish to determine whether a control is active, you should call `IsControlActive` instead of testing the `contrlHilite` field of the control structure.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

HiliteControl

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

Call `ActivateControl` (page 135) and `DeactivateControl` (page 136) instead of `HiliteControl` to activate or deactivate a control. This is important if the control is in an embedding hierarchy, since calling these functions will ensure that any latent embedded controls will be activated and deactivated correctly.

SendControlMessage

NEW WITH THE APPEARANCE MANAGER

Sends a message to a control definition function.

```
pascal SInt32 SendControlMessage (
                    ControlHandle inControl,
                    SInt16 inMessage,
                    SInt32 inParam);
```

`inControl` On input, a handle to the control to which you are sending a low-level message.

`inMessage` A bit field representing the message(s) you wish to send; see "Messages" (page 164).

`inParam` The message-dependent data passed in the `param` parameter of the control definition function.

function result Returns a signed 32-bit integer which contains varying data depending upon the message sent; see "Messages" (page 164).

DISCUSSION

Your application does not normally need to call the `SendControlMessage` function. If you have a special need to call a control definition function directly, call `SendControlMessage` to access and manipulate the control's attributes.

Before calling `SendControlMessage`, you should determine whether the control supports the specific message you wish to send by calling `GetControlFeatures` (page 158) and examining the feature bit field returned. If there are no feature bits returned that correspond to the message you wish to send (for messages 0 through 12), you can assume that all system controls support that message.

SEE ALSO

`MyControlDefProc` (page 163).

"Appearance Manager Gestalt Selector Constants" (page 21).

Displaying Controls

DrawOneControl

CHANGED WITH THE APPEARANCE MANAGER

Draws a single control and any embedded controls that are currently visible in the specified window.

```
pascal void DrawOneControl (ControlHandle theControl);
```

`theControl` On input, a handle to the control you want to draw.

DISCUSSION

Although you should generally use the function `UpdateControls` to update controls, you can use the `DrawOneControl` function to update a single control. If an embedding hierarchy exists and the control passed in has embedded controls, `DrawOneControl` draws the control and embedded controls. If the root control for a window is passed in, the result is the same as if `DrawControls` was called.

SEE ALSO

"Embedding Controls" (page 123).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`DrawOneControl` does not draw any embedded controls, because embedding hierarchies are not supported.

DrawControlInCurrentPort

NEW WITH THE APPEARANCE MANAGER

Draws a control in the current graphics port.

```
pascal void DrawControlInCurrentPort (ControlHandle inControl);
```

`inControl` On input, a handle to the control you wish to draw.

DISCUSSION

Normally, controls are automatically drawn in their owner's graphics port with `DrawControls`, `DrawOneControl` (page 139), and `UpdateControls`. `DrawControlInCurrentPort` permits easy offscreen control drawing and printing. All standard system controls support this function.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

Handling Events in Controls

FindControlUnderMouse

NEW WITH THE APPEARANCE MANAGER

Determines whether a mouse-down event has occurred in a control and, if so, in which.

```
pascal ControlHandle FindControlUnderMouse (
                     Point inWhere,
                     WindowPtr inWindow,
                     SInt16 *outPart);
```

`inWhere` On input, a point, specified in coordinates local to the window, where the mouse-down event occurred. Before calling `FindControlUnderMouse`, use the QuickDraw `GlobalToLocal` function to convert the point stored in the `where` field of the event structure (which describes the location of the mouse-down event) to coordinates local to the window.

`inWindow` On input, a pointer to the window in which the mouse-down event occurred.

`outPart` On output, a pointer to the part code of the control part which was selected; see "Control Part Code Constants" (page 99).

function result Returns a handle to the control that was selected. If the mouse-down event did not occur over a control part, `nil` is returned.

DISCUSSION

You should call the `FindControlUnderMouse` function instead of `FindControl` (page 142) to determine whether a mouse-down event occurred in a control, particularly if an embedding hierarchy is present. `FindControlUnderMouse` will return a handle to the control even if no part was hit and can determine whether a mouse-down event has occurred even if the control is deactivated, while `FindControl` does not.

When a mouse-down event occurs, your application should call `FindControlUnderMouse` after using the Window Manager function `FindWindow` to ascertain that a mouse-down event has occurred in the content region of a window containing controls.

SEE ALSO

"Embedding Controls" (page 123).

"Appearance Manager Gestalt Selector Constants" (page 21).

FindControl

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, you should call `FindControlUnderMouse` (page 141) to determine whether a mouse-down event occurred in a control. `FindControlUnderMouse` will return a handle to the control even if no part was hit and can determine whether a mouse-down event has occurred even if the control is deactivated, while `FindControl` does not.

HandleControlKey

NEW WITH THE APPEARANCE MANAGER

Sends a keyboard event to a control with keyboard focus.

```
pascal SInt16 HandleControlKey (
                    ControlHandle inControl,
                    SInt16 inKeyCode,
                    SInt16 inCharCode,
                    SInt16 inModifiers);
```

`inControl` On input, a handle to the control that currently has keyboard focus.

`inKeyCode`	The virtual key code, derived from the event structure. This value represents the key pressed or released by the user. It is always the same for a specific physical key on a particular keyboard regardless of which modifier keys were also pressed.
`inCharCode`	A particular character, derived from the event structure. The value that is generated depends on the virtual key code, the state of the modifier keys, and the current `'KCHR'` resource.
`inModifiers`	A constant from the `modifiers` field of the event structure specifying the state of the modifier keys and the mouse button at the time the event was posted.
function result	Returns the part code that was hit during the keyboard event; see "Control Part Code Constants" (page 99).

DISCUSSION

If you have determined that a keyboard event has occurred in a given window, before calling the `HandleControlKey` function, call `GetKeyboardFocus` (page 148) to get the handle to the control that currently has keyboard focus. The `HandleControlKey` function passes the values specified in its `inKeyCode`, `inCharCode`, and `inModifiers` parameters to control definition functions that set the `kControlSupportsFocus` feature bit.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

IdleControls

NEW WITH THE APPEARANCE MANAGER

Performs idle event processing.

```
pascal void IdleControls (WindowPtr inWindow);
```

`inWindow`	On input, a pointer to the window that contains controls which support idle events.

DISCUSSION

Your application should call the `IdleControls` function to give idle time to any controls that want the `kControlMsgIdle` message. `IdleControls` calls the control with an idle event so the control can do idle-time processing. You should call `IdleControls` at least once in your event loop. See "Performing Idle Processing" (page 181) for more details on how a control definition function should handle idle processing.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

HandleControlClick

NEW WITH THE APPEARANCE MANAGER

Follows and responds to cursor movements in a control while the mouse button is down and determines the control part in which the next mouse-up event occurs.

```
pascal ControlPartCode HandleControlClick (
                    ControlHandle inControl,
                    Point inWhere,
                    SInt16 inModifiers,
                    ControlActionUPP inAction);
```

`inControl` On input, a handle to the control in which the mouse-down event occurred. Pass the control handle returned by `FindControl` or `FindControlUnderMouse`.

`inWhere` A point, specified in local coordinates, where the mouse-down event occurred. Supply the same point you passed to `FindControl` or `FindControlUnderMouse`.

`inModifiers` Information about the state of the modifier keys and the mouse button at the time the event was posted.

`inAction` On input, a pointer to an action function defining what action your application takes while the user holds down the mouse button. The value of the `inAction` parameter can be a valid `procPtr`, `nil`, or -1. A value of -1 indicates that the control

should either perform auto tracking, or if it is incapable of doing so, do nothing (like `nil`). For custom controls, what you pass in this parameter depends on how you define the control. If the part index is greater than 128, the pointer must be of type `DragGrayRegionUPP` unless the control supports live feedback, in which case it should be a `ControlActionUPP`.

function result Returns a value of type `ControlPartCode` identifying the control's part; see "Control Part Code Constants" (page 99).

DISCUSSION

Call the `HandleControlClick` function after a call to `FindControl` or `FindControlUnderMouse`. The `HandleControlClick` function should be called instead of `TrackControl` (page 146) to follow the user's cursor movements in a control and provide visual feedback until the user releases the mouse button. Unlike `TrackControl`, `HandleControlClick` allows modifier keys to be passed in so that the control may use these if the control (such as a list box or editable text field) is set up to handle its own tracking.

The visual feedback given by `HandleControlClick` depends on the control part in which the mouse-down event occurs. When highlighting is appropriate, for example, `HandleControlClick` highlights the control part (and removes the highlighting when the user releases the mouse button). When the user holds down the mouse button while the cursor is in an indicator (such as the scroll box of a scroll bar) and moves the mouse, `HandleControlClick` responds by dragging a dotted outline or a ghost image of the indicator. If the user releases the mouse button when the cursor is in an indicator such as the scroll box, `HandleControlClick` calls the control definition function to reposition the indicator.

While the user holds down the mouse button with the cursor in one of the standard controls, `HandleControlClick` performs the following actions, depending on the value you pass in the parameter `inAction`.

- If you pass `nil` in the `inAction` parameter, `HandleControlClick` uses no action function and therefore performs no additional actions beyond highlighting the control or dragging the indicator. This is appropriate for push buttons, checkboxes, radio buttons, and the scroll box of a scroll bar.
- If you pass a pointer to an action function in the `inAction` parameter, it must define some action that your application repeats as long as the user holds

down the mouse button. This is appropriate for the scroll arrows and gray areas of a scroll bar.

- If you pass `(ControlActionUPP)-1L` in the `inAction` parameter, `HandleControlClick` looks in the `contrlAction` field of the control structure for a pointer to the control's action function. This is appropriate when you are tracking the cursor in a pop-up menu. You can call `GetControlAction` to determine the value of this field, and you can call `SetControlAction` (page 153) to change this value. If the `contrlAction` field of the control structure contains a function pointer, `HandleControlClick` uses the action function it points to; if the field of the control structure also contains the value `(ControlActionUPP)-1L`, `HandleControlClick` calls the control definition function to perform the necessary action; you may wish to do this if you define your own control definition function for a custom control. If the field of the control structure contains the value `nil`, `HandleControlClick` performs no action.

Note
For `'CDEF'` resources that implement custom dragging, you usually call `HandleControlClick`, which returns 0 regardless of the user's changes of the control setting. To avoid this, you should use another method to determine whether the user has changed the control setting, for instance, comparing the control's value before and after your call to `HandleControlClick`.

SEE ALSO

`MyActionProc` (page 185).

"Appearance Manager Gestalt Selector Constants" (page 21).

TrackControl

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, call `HandleControlClick` (page 144) instead of `TrackControl` to follow the user's cursor movements in a control and provide visual feedback until the user releases the mouse button.

`HandleControlClick` allows modifier keys to be passed in so that the control may use these if the control is set up to handle its own tracking.

Handling Keyboard Focus

A control with **keyboard focus** receives keyboard events. The Dialog Manager tests to see which control has keyboard focus when a keyboard event is processed and sends the event to that control. If no control has keyboard focus, the keyboard event is discarded. Currently, the list box, clock, and editable text controls are the only standard system controls that support keyboard focus. A control retains keyboard focus if it is hidden or deactivated.

A **focus ring** is drawn around the control with keyboard focus. When creating your own controls, allow space for the focus ring. For more details on designing with focus rings, see *Mac OS 8 Human Interface Guidelines*.

Keyboard focus is only available if an embedding hierarchy has been established in the focusable control's window. The default focusing order is based on the order in which controls are added to the window. For more details on embedding hierarchies, see "Embedding Controls" (page 123).

SetKeyboardFocus

NEW WITH THE APPEARANCE MANAGER

Sets the current keyboard focus to a specified control part for a specified window.

```
pascal OSErr SetKeyboardFocus (
                    WindowPtr inWindow,
                    ControlHandle inControl,
                    ControlFocusPart inPart);
```

`inWindow` On input, a pointer to the window containing the control that you wish to receive keyboard focus.

`inControl` On input, a handle to the control that you wish to receive keyboard focus.

`inPart`	A part code specifying the part of a control which will receive keyboard focus. To clear a control's keyboard focus, pass `kControlFocusNoPart`. See "Handling Keyboard Focus" (page 178).
function result	A result code; see "Result Codes" (page 110). The result code `errNoRootControl` indicates that keyboard focus is unavailable because the window does not have an embedding hierarchy established. If the specified control doesn't support keyboard focus, `errControlDoesntSupportFocus` is returned

DISCUSSION

The `SetKeyboardFocus` function sets the keyboard focus to a specified control part. The control to receive keyboard focus can be deactivated or invisible. This permits you to set the focus for an item in a dialog box before the dialog box is displayed.

SEE ALSO

`GetKeyboardFocus` (page 148).

"Handling Keyboard Focus" (page 147).

"Appearance Manager Gestalt Selector Constants" (page 21).

GetKeyboardFocus

NEW WITH THE APPEARANCE MANAGER

Gets a handle to the control with the current keyboard focus for a specified window.

```
pascal OSErr GetKeyboardFocus (
                    WindowPtr inWindow,
                    ControlHandle* outControl);
```

`inWindow`	On input, a pointer to the window for which you wish to get keyboard focus.

`outControl`	On output, a pointer to a handle to the control that currently has keyboard focus. Produces `nil` if no control has focus.
function result	A result code; see "Result Codes" (page 110). The result code `errNoRootControl` indicates that keyboard focus is unavailable because the window does not have an embedding hierarchy established.

DISCUSSION

The `GetKeyboardFocus` function returns the handle of the control with current keyboard focus within a specified window.

SEE ALSO

`SetKeyboardFocus` (page 147).

"Handling Keyboard Focus" (page 147).

"Appearance Manager Gestalt Selector Constants" (page 21).

AdvanceKeyboardFocus

NEW WITH THE APPEARANCE MANAGER

Advances the keyboard focus to the next focusable control in the window.

```
pascal OSErr AdvanceKeyboardFocus (WindowPtr inWindow);
```

`inWindow`	On input, a pointer to the window for which you wish to advance keyboard focus.
function result	A result code; see "Result Codes" (page 110). The result code `errNoRootControl` indicates that keyboard focus is unavailable because the window does not have an embedding hierarchy established.

DISCUSSION

The `AdvanceKeyboardFocus` function skips over deactivated and hidden controls until it finds the next focusable control in the window. If it does not find a focusable item, it simply returns.

When `AdvanceKeyboardFocus` is called, the Control Manager calls your control definition function and passes `kControlMsgFocus` in its `message` parameter and `kControlFocusNextPart` in its `param` parameter. In response to this message, your control definition function should change keyboard focus to its next part, the entire control, or remove keyboard focus from the control, depending upon the circumstances. See "Handling Keyboard Focus" (page 178) for a discussion of possible responses to this message.

SEE ALSO

`ReverseKeyboardFocus` (page 150).

"Handling Keyboard Focus" (page 147).

"Appearance Manager Gestalt Selector Constants" (page 21).

ReverseKeyboardFocus

NEW WITH THE APPEARANCE MANAGER

Reverses the progression of keyboard focus, returning focus to the previous control to receive keyboard focus in the window.

```
pascal OSErr ReverseKeyboardFocus (WindowPtr inWindow);
```

`inWindow` On input, a pointer to the window for which you wish to reverse keyboard focus.

function result A result code; see "Result Codes" (page 110). The result code `errNoRootControl` indicates that keyboard focus is unavailable because the window does not have an embedding hierarchy established.

DISCUSSION

The `ReverseKeyboardFocus` function skips over deactivated and hidden controls until it finds the previous control to receive keyboard focus in the window.

When `ReverseKeyboardFocus` is called, the Control Manager calls your control definition function and passes `kControlMsgFocus` in its `message` parameter and `kControlFocusPrevPart` in its `param` parameter. In response to this message, your control definition function should change keyboard focus to its previous part, the entire control, or remove keyboard focus from the control, depending upon the circumstances. See "Handling Keyboard Focus" (page 178) for a discussion of possible responses to this message.

SEE ALSO

`AdvanceKeyboardFocus` (page 149).

"Handling Keyboard Focus" (page 147).

"Appearance Manager Gestalt Selector Constants" (page 21).

ClearKeyboardFocus

NEW WITH THE APPEARANCE MANAGER

Clears the keyboard focus for the currently focused control in the specified window.

```
pascal OSErr ClearKeyboardFocus (WindowPtr inWindow);
```

`inWindow` On input, a pointer to the window in which you wish to clear keyboard focus.

function result A result code; see "Result Codes" (page 110). The result code `errNoRootControl` indicates that keyboard focus is unavailable because the window does not have an embedding hierarchy established.

DISCUSSION

When the `ClearKeyboardFocus` function is called, the Control Manager calls your control definition function and passes `kControlMsgFocus` in its `message`

parameter and `kControlFocusNoPart` in its `param` parameter. See "Handling Keyboard Focus" (page 178) for a discussion of possible responses to this message.

SEE ALSO

"Handling Keyboard Focus" (page 147).

"Appearance Manager Gestalt Selector Constants" (page 21).

Accessing and Changing Control Settings and Data

GetBestControlRect

NEW WITH THE APPEARANCE MANAGER

Determines a control's optimal size and text placement.

```
pascal OSErr GetBestControlRect (
                    ControlHandle inControl,
                    Rect *outRect,
                    SInt16 *outBaseLineOffset);
```

`inControl` On input, a handle to the control whose size you wish to determine.

`outRect` On input, a pointer to the control rectangle you wish to modify; pass an empty rectangle (0, 0, 0, 0). On output, a pointer to the rectangle that the control has determined to be optimal. If the control doesn't support getting an optimal size rectangle, the control's bounding rectangle is passed back.

`outBaseLineOffset`
On output, the offset from the bottom of control to the base of the text (usually a negative value). If the control doesn't support optimal sizing or has no text, 0 is passed back.

function result A result code; see "Result Codes" (page 110).

DISCUSSION

The `GetBestControlRect` function should be called to automatically position and size controls in accordance with human interface guidelines. This function is particularly helpful in determining the correct placement of control text whose length is not known until run-time. For example, `StandardAlert` (page 261) uses `GetBestControlRect` to automatically size and position buttons in a newly created alert box.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

SetControlAction

CHANGED WITH THE APPEARANCE MANAGER

Sets or changes the action function for the specified control's control structure.

```
pascal void SetControlAction (
                    ControlHandle theControl,
                    ControlActionUPP actionProc);
```

`theControl` On input, a handle to the control whose action function you wish to change.

`actionProc` On input, a pointer to an action function defining what action your application takes while the user holds down the mouse button.

DISCUSSION

The `SetControlAction` function changes the `contrlAction` field of the control structure to point to the action function specified in the `actionProc` parameter. If the cursor is in the specified control, `HandleControlClick` (page 144) or `TrackControl` (page 146) call this action function when the user holds down the mouse button. You must provide the action function, and it must define some action to perform repeatedly as long as the user holds down the mouse button. `HandleControlUnderClick` and `TrackControl` always highlight and drag the control as appropriate.

Note
`SetControlAction` should be used to set the application-defined action function for providing live feedback for standard system scroll bar controls.

SEE ALSO

`MyActionProc` (page 185).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Live feedback is not supported.

SetControlColor

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard controls, colors are determined by the current theme. If you are creating your own control definition function, you can still set your own colors with the `SetControlColor` function.

SetControlData

NEW WITH THE APPEARANCE MANAGER

Sets control-specific data.

```
pascal OSErr SetControlData (
                    ControlHandle inControl,
                    ControlPartCode inPart,
                    ResType inTagName,
                    Size inSize,
                    Ptr inData);
```

`inControl` On input, a handle to the control.

`inPart`	The part code of the control part whose control-specific data you wish to set; see "Control Part Code Constants" (page 99). Passing `kControlEntireControl` indicates that either the control has no parts or the data is not tied to any specific part of the control.
`inTagName`	A constant representing the control-specific data you wish to set; see "Control Data Tag Constants" (page 83).
`inSize`	The size (in bytes) of the data pointed to by the `inData` parameter. For variable-length control data, pass the value returned in the `outMaxSize` parameter of `GetControlDataSize` (page 157) in the `inSize` parameter. The number of bytes must match the actual data size.
`inData`	On input, a pointer to a buffer allocated by your application. This buffer contains the data that you are sending to the control. After calling `SetControlData`, your application is responsible for disposing of this buffer, if necessary, as information is copied by control.
function result	A result code; see "Result Codes" (page 110). The result code `errDataNotSupported` indicates that the `inTagName` parameter is not valid.

DISCUSSION

The `SetControlData` function sets control-specific data represented by the value in the `inTagName` parameter to the data pointed to by the `inData` parameter. `SetControlData` could be used, for example, to switch a progress indicator from a determinate to indeterminate state. For a list of the control attributes that can be set, see "Control Data Tag Constants" (page 83).

SEE ALSO

`GetControlData` (page 156).

"Appearance Manager Gestalt Selector Constants" (page 21).

GetControlData

NEW WITH THE APPEARANCE MANAGER

Gets control-specific data.

```
pascal OSErr GetControlData (
                    ControlHandle inControl,
                    ControlPartCode inPart,
                    ResType inTagName,
                    Size inBufferSize,
                    Ptr inBuffer,
                    Size *outActualSize);
```

`inControl` On input, a handle to the specified control.

`inPart` The part code of the control part whose control-specific data you wish to set; see "Control Part Code Constants" (page 99). Passing `kControlEntireControl` indicates that either the control has no parts or the data is not tied to any specific part of the control.

`inTagName` A constant representing the control-specific data you wish to get; see "Control Data Tag Constants" (page 83).

`inBufferSize` The size (in bytes) of the data pointed to by the `inBuffer` parameter. For variable-length control data, pass the value returned in the `outMaxSize` parameter of `GetControlDataSize` (page 157) in the `inBufferSize` parameter. The number of bytes must match the actual data size.

`inBuffer` On input, a pointer to a buffer allocated by your application. On output, a copy of the control-specific data. If you pass `nil` on input, it is equivalent to calling `GetControlDataSize` (page 157). The actual size of the control-specific data will be returned in the `outActualSize` parameter. For variable-length data, the number of bytes must match the actual data size.

`outActualSize` On output, a pointer to the actual size of the data.

function result A result code; see "Result Codes" (page 110). The result code `errDataNotSupported` indicates that the `inTagName` parameter is not valid. The result code `errDataSizeMismatch` indicates that the value in the `inBufferSize` parameter does not match the value in the `outActualSize` parameter.

DISCUSSION

The `GetControlData` function will only copy the amount of data specified in the `inBufferSize` parameter, but will tell you the actual size of the buffer so you will know if the data was truncated.

SEE ALSO

`SetControlData` (page 154).

"Appearance Manager Gestalt Selector Constants" (page 21).

GetControlDataSize

NEW WITH THE APPEARANCE MANAGER

Returns the size (in bytes) of a control's tagged data.

```
pascal OSErr GetControlDataSize (
                    ControlHandle inControl,
                    ControlPartCode inPart,
                    ResType inTagName,
                    Size *outMaxSize);
```

`inControl`	On input, a handle to the specified control.
`inPart`	The part code of the control part whose control-specific data you wish to set; see "Control Part Code Constants" (page 99). Passing `kControlEntireControl` indicates that either the control has no parts or the data is not tied to any specific part of the control.
`inTagName`	A constant representing the control-specific data you wish to get; see "Control Data Tag Constants" (page 83).
`outMaxSize`	On output, a pointer to the size of the control's tagged data. This value should be passed to `SetControlData` (page 154) and `GetControlData` (page 156) to allocate a sufficiently large buffer for variable-length data.
function result	A result code; see "Result Codes" (page 110). The result code `errDataNotSupported` indicates that the `inTagName` parameter is not valid.

DISCUSSION

Pass the value returned in the `outMaxSize` parameter of `GetControlDataSize` in the `inBufferSize` parameter of `SetControlData` (page 154) and `GetControlData` (page 156) to allocate an adequate buffer for variable-length data.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

GetControlFeatures

NEW WITH THE APPEARANCE MANAGER

Returns the Appearance-compliant features a control supports.

```
pascal OSErr GetControlFeatures (
                    ControlHandle inControl,
                    UInt32 *outFeatures);
```

`inControl` On input, a handle to the control whose features you wish to determine.

`outFeatures` On output, a pointer to a bit field specifying the features the control supports. For a list of the features a control may support, see "Specifying Which Appearance-Compliant Messages Are Supported" (page 174).

function result A result code; see "Result Codes" (page 110). The result code `errMsgNotSupported` indicates that the control does not support Appearance-compliant features.

DISCUSSION

The `GetControlFeatures` function returns the features a control definition function supports, in response to a `kControlMsgGetFeatures` message.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

SetControlFontStyle

NEW WITH THE APPEARANCE MANAGER

Sets the font style for the specified control.

```
pascal OSErr SetControlFontStyle (
                    ControlHandle inControl,
                    const ControlFontStyleRec *inStyle);
```

`inControl` On input, a handle to the control whose font style you wish to set.

`inStyle` On input, a pointer to a control font style structure (page 103). If the `flags` field is cleared, the control uses the system font unless the control variant `kControlUsesOwningWindowsFontVariant` has been specified (control uses window font).

function result A result code; see "Result Codes" (page 110).

DISCUSSION

The `SetControlFontStyle` function sets the font style for a given control. To specify the font for controls in a dialog box, it is generally easier to use the dialog control font table resource (page 254). `SetControlFontStyle` allows you to override a control's default font (system or window font, depending upon whether the control variant `kControlUsesOwningWindowsFontVariant` has been specified). Once you have set a control's font with this function, you can cause the control to revert to its default font by passing a control font style structure with a cleared `flags` field in the `inStyle` parameter.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

SetControlVisibility

NEW WITH THE APPEARANCE MANAGER

Sets the visibility of a control, and any embedded controls, and specifies whether it will be drawn.

```
pascal OSErr SetControlVisibility (
                    ControlHandle inControl,
                    Boolean inIsVisible,
                    Boolean inDoDraw);
```

`inControl` On input, a handle to the control.

`inIsVisible` A Boolean value indicating whether the control is visible or invisible. If this value is set to `true`, the control will be visible. If `false`, the control will be invisible. If you wish to show a control (and latent embedded subcontrols) but do not want to cause screen drawing, pass `true` for this parameter and false in the `inDoDraw` parameter.

`inDoDraw` On input, Boolean value indicating whether the control should be drawn or erased. If `true`, the control's display on the screen should be updated (drawn or erased) based on the value passed in the `inIsVisible` parameter. If `false`, the display will not be updated.

function result A result code; see "Result Codes" (page 110).

DISCUSSION

You should call the `SetControlVisibility` function instead of setting the `contrlVis` field of the control structure to set the visibility of a control and specify whether it will be drawn. If the control has embedded controls, `SetControlVisibility` allows you to set their visibility and specify whether or not they will be drawn. If you wish to show a control but do not want it to be drawn onscreen, pass `true` in the `inIsVisible` parameter and false in the `inDoDraw` parameter.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

IsControlVisible

NEW WITH THE APPEARANCE MANAGER

Indicates whether a control is visible.

```
pascal Boolean IsControlVisible (ControlHandle inControl);
```

`inControl` On input, a handle to the control whose visibility you wish to determine.

function result Returns a Boolean value. If `true`, the control is visible. If `false`, the control is hidden.

DISCUSSION

If you wish to determine whether a control is visible, call `IsControlVisible` instead of testing the `contrlVis` field of the control structure.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

SetUpControlBackground

NEW WITH THE APPEARANCE MANAGER

Sets the background for a control.

```
pascal OSErr SetUpControlBackground (
                    ControlHandle inControl,
                    SInt16 inDepth,
                    Boolean inIsColorDevice);
```

`inControl` On input, a handle to the control whose background you wish to set.

`inDepth` The bit depth (in pixels) of the current graphics port.

`inIsColorDevice`
A Boolean value. Set to `true` to indicate that you are drawing on a color device; set to `false` for a monochrome device.

function result A result code; see "Result Codes" (page 110).

DISCUSSION

The `SetUpControlBackground` function allows you to set the background of a control. This function is typically called by control definition functions that are embedded in other controls. You might call `SetUpControlBackground` in response to an application-defined function installed in a user pane control; see "Defining Your Own User Pane Functions" (page 189). `SetUpControlBackground` ensures that the background color is always correct when calling `EraseRect` and `EraseRgn`. If your control spans multiple monitors, `SetUpControlBackground` should be called for each device that your control is drawing on; see "Graphics Devices" in *Imaging With QuickDraw* for more details on handling device loops.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

Defining Your Own Control Definition Function

A control definition function determines how a control generally looks and behaves. Various Control Manager functions call a control definition function whenever they need to perform a control-dependent action, such as drawing the control on the screen. In addition to standard control definition functions, defined by the system, you can make your own custom control definition functions.

The Control Manager calls the Resource Manager to access a control definition function with the given resource ID; for a description of how to derive a control definition function ID, see "Control Definition IDs" (page 71). The Resource Manager reads a control definition function into memory and returns a handle to it. The Control Manager stores this handle in the `contrlDefProc` field of the control structure.

When various Control Manager functions need to perform a type-dependent action on the control, they call the control definition function and pass it the variation code for its type as a parameter. You can define your own variation codes; this allows you to use one `'CDEF'` resource to handle several variations of the same general control. See "The Control Definition Function Resource" (page 113) and "The Control Resource" (page 111) for further discussion of controls, their resources, and their IDs.

If you choose to provide your own control definition functions, these functions should apply the user's desktop color choices the same way the standard control definition functions do. You can use control color tables of any desired size and define their contents in any way you wish, except that part indices and messages 0 through 127 are reserved for system definition.

MyControlDefProc

CHANGED WITH THE APPEARANCE MANAGER

If you wish to define new, nonstandard controls for your application, you must write a control definition function and store it in a resource file as a resource of type `'CDEF'`.

The Control Manager declares the type for an application-defined control definition function as follows:

```
typedef pascal SInt32 (*ControlDefProcPtr)
                           (SInt16 varCode,
                            ControlHandle theControl,
                            ControlDefProcMessage message,
                            SInt32 param);
```

The Control Manager defines the data type `ControlDefUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlDefUPP;
```

You typically use the `NewControlDefProc` macro like this:

```
ControlDefUPP myControlDefUPP;
myControlDefUPP = NewControlDefProc (MyControl);
```

You typically use the `CallControlDefProc` macro like this:

```
CallControlDefProc(myControlDefUPP, varCode, theControl, message, param);
```

Here's how to declare the function `MyControlDefProc`:

```
pascal SInt32 MyControlDefProc (
                    SInt16 varCode,
                    ControlHandle theControl,
                    ControlDefProcMessage message,
                    SInt32 param);
```

`varCode`	The control's variation code.
`theControl`	A handle to the control that the operation will affect.
`message`	A code for the task to be performed. The `message` parameter contains one of the task codes defined in "Messages" (page 164). The subsections that follow explain each of these tasks in detail.
`param`	Data associated with the task specified by the `message` parameter. If the task requires no data, this parameter is ignored.
function result	The function results that your control definition function returns depend on the value that the Control Manager passes in the `message` parameter.

DISCUSSION

The Control Manager calls your control definition function under various circumstances; the Control Manager uses the `message` parameter to inform your control definition function what action it must perform. The data that the Control Manager passes in the `param` parameter, the action that your control definition function must undertake, and the function results that your control definition function returns all depend on the value that the Control Manager passes in the `message` parameter. The rest of this section describes how to respond to the various values that the Control Manager passes in the `message` parameter.

Messages

The Control Manager passes constants of type `ControlDefProcMessage` to indicate the action your control definition function must perform.

```
enum {
    drawCntl                    = 0,
    testCntl                    = 1,
    calcCRgns                   = 2,
    initCntl                    = 3,
    dispCntl                    = 4,
    posCntl                     = 5,
    thumbCntl                   = 6,
    dragCntl                    = 7,
    autoTrack                   = 8,
    calcCntlRgn                 = 10,
    calcThumbRgn                = 11,
    kControlMsgDrawGhost        = 13,
    kControlMsgCalcBestRect     = 14,
    kControlMsgHandleTracking   = 15,
    kControlMsgFocus            = 16,
    kControlMsgKeyDown          = 17,
    kControlMsgIdle             = 18,
    kControlMsgGetFeatures      = 19,
    kControlMsgSetData          = 20,
    kControlMsgGetData          = 21,
    kControlMsgActivate         = 22,
    kControlMsgSetUpBackground  = 23,
    kControlMsgSubValueChanged  = 25,
    kControlMsgCalcValueFromPos = 26,
    kControlMsgTestNewMsgSupport= 27,
    kControlMsgSubControlAdded  = 28,
    kControlMsgSubControlRemoved= 29
};
typedef SInt16 ControlDefProcMessage;
```

Constant descriptions

`drawCntl`	Draw the entire control or part of a control.
`testCntl`	Test where the mouse has been pressed.
`calcCRgns`	Calculate the region for the control or the indicator in 24-bit systems. This message is obsolete in Mac OS 7.6 and later.
`initCntl`	Perform additional control initialization.
`dispCntl`	Perform additional control disposal actions.

`posCntl` Move and update the indicator setting.

`thumbCntl` Calculate the parameters for dragging the indicator.

`dragCntl` Perform customized dragging (of the control or its indicator).

`autoTrack` Execute the specified action function.

`calcCntlRgn` Calculate the control region in 32-bit systems.

`calcThumbRgn` Calculate the indicator region in 32-bit systems.

`kControlMsgDrawGhost`
Draw a ghost image of the indicator.

`kControlMsgCalcBestRect`
Calculate the optimal control rectangle.

`kControlMsgHandleTracking`
Perform custom tracking.

`kControlMsgFocus` Handle keyboard focus.

`kControlMsgKeyDown` Handle keyboard events.

`kControlMsgIdle` Perform idle processing.

`kControlMsgGetFeatures`
Specify which Appearance-compliant messages are supported.

`kControlMsgSetData`
Set control-specific data.

`kControlMsgGetData`
Get control-specific data.

`kControlMsgActivate`
Handle activate and deactivate events.

`kControlMsgSetUpBackground`
Set the control's background color or pattern (only available if the control supports embedding).

`kControlMsgSubValueChanged`
Be informed that the value of a subcontrol embedded in the control has changed; this message is useful for radio groups. Only available with Appearance 1.0.1 and later.

`kControlMsgCalcValueFromPos`
Support live feedback while dragging the indicator and calculate the control value based on the new indicator region.

`kControlMsgTestNewMsgSupport`
Specify whether Appearance-compliant messages are supported.

`kControlMsgSubControlAdded`
Be informed that a subcontrol has been embedded in the control. This message is only available with Appearance 1.0.1 and later.

`kControlMsgSubControlRemoved`
Be informed that a subcontrol is about to be removed from the control. This message is only available with Appearance 1.0.1 and later.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Only the following messages will be sent to your control definition function:

```
drawCntl                = 0,
testCntl                = 1,
calcCRgns               = 2,
initCntl                = 3,
dispCntl                = 4,
posCntl                 = 5,
thumbCntl               = 6,
dragCntl                = 7,
autoTrack               = 8,
calcCntlRgn             = 10,
calcThumbRgn            = 11
```

Drawing the Control or Its Part

When the Control Manager passes the value `drawCntl` in the `message` parameter, your control definition function should respond by drawing the indicator or the entire control.

The Control Manager passes one of the following drawing constants in the low word of the `param` parameter to specify whether the user is drawing an indicator or the whole control. The high-order word of the `param` parameter may contain undefined data; therefore, evaluate only the low-order word of this parameter.

```
enum {
    kDrawControlEntireControl       = 0,
    kDrawControlIndicatorOnly       = 129
};
```

Constant descriptions

`kDrawControlEntireControl`
Draw the entire control.

`kDrawControlIndicatorOnly`
Draw the indicator only.

With the exception of part code 128, which is reserved for future use and should not be used, any other value indicates a part code for the control.

If the specified control is visible, your control definition function should draw the control (or the part specified in the `param` parameter) within the control's rectangle. If the control is invisible (that is, if its `contrlVis` field is set to 0), your control definition function does nothing.

When drawing the control or its part, take into account the current values of its `contrlHilite` and `contrlValue` fields in the control structure.

If the part code for your control's indicator is passed in `param`, assume that the indicator hasn't moved; the Control Manager, for example, may be calling your control definition function so that you may simply highlight the indicator. However, when your application calls `SetControlValue`, `SetControlMinimum`, and `SetControlMaximum`, they in turn may call your control definition function with the `drawCntl` message to redraw the indicator. Since these functions have no way of determining what part code you chose for your indicator, they all pass 129 in `param`, meaning that you should move your indicator. Your control definition function must detect this part code as a special case and remove the indicator from its former location before drawing it. If your control has more than one indicator, you should interpret 129 to mean all indicators.

When sent the message `drawCntl`, your control definition function should return 0 as its function result.

Testing Where the Mouse-Down Event Occurs

When the Control Manager passes the value for the `testCntl` constant in the `message` parameter, your control definition function should respond by determining whether a specified point is in a visible control.

The Control Manager passes a point (in local coordinates) in the `param` parameter. The point's vertical coordinate is contained in the high-order word of the long integer, and horizontal coordinate is contained in the low-order word.

Your control definition function should return the part code of the part that contains the specified point; it should return 0 if the point is outside the control or if the control is inactive.

Calculating the Control and Indicator Regions on 24-Bit Systems

When the Control Manager passes the value for the `calcCRgns` constant in the `message` parameter, your control definition function should calculate the region passed in the `param` parameter for the specified control or its indicator.

The Control Manager passes a QuickDraw region handle in the `param` parameter. If the high-order bit of `param` is set, the region requested is that of the control's indicator; otherwise, the region requested is that of the entire control. Your control definition function should clear the high bit of the region handle before calculating the region.

When passed this message, your control definition function should always return 0, and it should express the region in the local coordinate system of the control's window.

IMPORTANT

The `calcCRgns` message will never be sent to any system running on 32-bit mode and is therefore obsolete in Mac OS 7.6 and later. The `calcCntlRgn` and `calcThumbRgn` messages will be sent instead.

Calculating the Control and Indicator Regions on 32-Bit Systems

When the Control Manager passes the values for the `calcCntlRgn` or `calcThumbRgn` constants in the `message` parameter, your control definition function should calculate the region for the specified control or its indicator using the QuickDraw region handle passed in the `param` parameter .

If the Control Manager passes the value for the `calcThumbRgn` constant in the `message` parameter, calculate the region occupied by the indicator. If the Control Manager passes the value for the `calcCntlRgn` constant in the `message` parameter, calculate the region for the entire control.

When passed this message, your control definition function should always return 0, and it should express the region in the local coordinate system of the control's window.

Performing Additional Control Initialization

After initializing fields of a control structure as appropriate when creating a new control, the Control Manager passes `initCntl` in the `message` parameter to give your control definition function the opportunity to perform any type-specific initialization you may require. For example, the standard control definition function for scroll bars allocates space for a region to hold the scroll box and stores the region handle in the `contrlData` field of the new control structure.

When passed the value for the `initCntl` constant in the `message` parameter, your control definition function should ignore the `param` parameter and return 0 as a function result.

Performing Additional Control Disposal Actions

The function `DisposeControl` (page 121) passes `dispCntl` in the `message` parameter to give your control definition function the opportunity to carry out any additional actions when disposing of a control. For example, the standard definition function for scroll bars releases the memory occupied by the scroll box region, whose handle is kept in the `contrlData` field of the control structure.

When passed the value for the `dispCntl` constant in the `message` parameter, your control definition function should ignore the `param` parameter and return 0 as a function result.

Dragging the Control or Its Indicator

When a mouse-up event occurs in the indicator of a control, the `HandleControlClick` (page 144) or `TrackControl` (page 146) functions call your control definition function and pass `posCntl` in the `message` parameter. In this case, the Control Manager passes a point (in coordinates local to the control's window) in the `param` parameter that specifies the vertical and horizontal offset, in pixels, by which your control definition function should move the indicator from its current position. Typically, this is the offset between the points where the cursor was when the user pressed and released the mouse button while dragging the indicator. The point's vertical offset is contained in the high-order

word of the `param` parameter, and its horizontal offset is contained in the low-order word.

Your definition function should calculate the control's new setting based on the given offset and then, to reflect the new setting, redraw the control and update the `contrlValue` field in the control structure. Your control definition function should ignore the `param` parameter and return 0 as a function result.

Calculating Parameters for Dragging the Indicator

When the Control Manager passes the value for `thumbCntl` in the `message` parameter, your control definition function should respond by calculating values analogous to the `limitRect`, `slopRect`, and `axis` parameters of `DragControl` that constrain how the indicator is dragged. On entry, the fields `param->limitRect.top` and `param->limitRect.left` contain the point where the mouse-down event first occurred.

The Control Manager passes a pointer to a structure of type `IndicatorDragConstraint` in the `param` parameter:

```
struct IndicatorDragConstraint {
    Rect                limitRect;
    Rect                slopRect;
    DragConstraint      axis;
};
typedef struct IndicatorDragConstraint IndicatorDragConstraint;
typedef IndicatorDragConstraint *IndicatorDragConstraintPtr;
typedef IndicatorDragConstraintPtr *IndicatorDragConstraintHandle;
```

Field descriptions

`limitRect` A pointer to a rectangle—whose coordinates should normally coincide with or be contained in the window's content region—delimiting the area in which the user can drag the control's outline.

`slopRect` A pointer to a rectangle that allows some extra space for the user to move the mouse while still constraining the control within the rectangle specified in the `limitRect` parameter.

`axis` The axis along which the user may drag the control's outline.

Your definition function should store the appropriate values into the fields of the structure pointed to by the `param` parameter; they're analogous to the similarly named parameters of the Window Manager function `DragGrayRgn`.

Your control definition function should return 0 as function result.

Performing Custom Dragging

When the Control Manager passes the value for the `dragCntl` constant in the `message` parameter, the `param` parameter typically contains a custom dragging constant with one of the following values to specify whether the user is dragging an indicator or the whole control:

```
enum {
    kDragControlEntireControl   = 0,
    kDragControlIndicator       = 1
};
```

Constant descriptions

`kDragControlEntireControl`
Dragging the entire control.

`kDragControlIndicator`
Dragging the indicator.

Note
When the Appearance Manager is present, the message `kControlMsgHandleTracking` should be sent instead of `dragCntl` to handle any custom tracking; see "Performing Custom Tracking" (page 177).

If you want to use the Control Manager's default method of dragging, which is to call `DragControl` to drag the control or the Window Manager function `DragGrayRgn` to drag its indicator, return 0 as the function result for your control definition function.

If your control definition function returns a nonzero value, your control definition function (not the Control Manager) must drag the specified control (or its indicator) to follow the cursor until the user releases the mouse button. If the user drags the entire control, your definition function should use the function `MoveControl` to reposition the control to its new location after the user releases the mouse button. If the user drags the indicator, your definition function must calculate the control's new setting (based on the pixel offset

between the points where the cursor was when the user pressed and released the mouse button while dragging the indicator) and then, to reflect the new setting, redraw the control and update the `contrlValue` field in the control structure. Note that, in this case, the functions `HandleControlClick` (page 144) and `TrackControl` (page 146) return 0 whether or not the user changes the indicator's position. Thus, you must determine whether the user has changed the control's setting by another method, for instance, by comparing the control's value before and after the call to `HandleControlClick`.

Executing an Action Function

The only way to specify actions in response to all mouse-down events in a control or its indicator is to define your own control definition function that specifies an action function. When you create the control, your control definition function must first respond to the `initCntl` message by storing `(ControlDefUPP)-1L` in the `contrlAction` field of the control structure. (The Control Manager sends the `initCntl` message to your control definition function after initializing the fields of a new control structure.) Then, when your application passes `(ControlActionUPP)-1L` in the `actionProc` parameter of `HandleControlClick` (page 144) or `TrackControl` (page 146), `HandleControlClick` calls your control definition function with the `autoTrack` message. The Control Manager passes the part code of the part where the mouse-down event occurs in the `param` parameter. Your control definition function should then use this information to respond as an action function would.

Note
For the `autoTrack` message, the high-order word of the `param` parameter may contain undefined data; therefore, evaluate only the low-order word of this parameter.

If the mouse-down event occurs in an indicator of a control that supports live feedback, your action function should take two parameters (a handle to the control and the part code of the control where the mouse-down event first occurred). This action function is the same one you would use to define actions to be performed in control part codes in response to a mouse-down event; see `MyActionProc` (page 185).

If the mouse-down event occurs in an indicator of a control that does not support live feedback, your action function should take no parameters, because the user may move the cursor outside the indicator while dragging it; see `MyIndicatorActionProc` (page 186).

Specifying Whether Appearance-Compliant Messages Are Supported

If your control definition function supports Appearance-compliant messages, it should return `kControlSupportsNewMessages` as a function result when the Control Manager passes `kControlMsgTestNewMsgSupport` in the `message` parameter.

```
enum{
    kControlSupportsNewMessages = ' ok '
};
```

Constant description

`kControlSupportsNewMessages`
The control definition function supports new messages introduced with Mac OS 8 and the Appearance Manager.

Specifying Which Appearance-Compliant Messages Are Supported

If your control definition function supports Appearance-compliant messages, it should return a bit field of the features it supports in response to the `kControlMsgGetFeatures` message. Your control definition function should ignore the `param` parameter.

The bit field returned by your control definition function should be composed of one or more of the following bits:

```
enum{
    kControlSupportsGhosting    = 1 << 0,
    kControlSupportsEmbedding   = 1 << 1,
    kControlSupportsFocus       = 1 << 2,
    kControlWantsIdle           = 1 << 3,
    kControlWantsActivate       = 1 << 4,
    kControlHandlesTracking     = 1 << 5,
    kControlSupportsDataAccess  = 1 << 6,
    kControlHasSpecialBackground= 1 << 7,
    kControlGetsFocusOnClick    = 1 << 8,
    kControlSupportsCalcBestRect= 1 << 9,
    kControlSupportsLiveFeedback= 1 << 10,
    kControlHasRadioBehavior    = 1 << 11
};
```

Constant descriptions

`kControlSupportsGhosting`
: If this bit (bit 0) is set, the control definition function supports the `kControlMsgDrawGhost` message.

`kControlSupportsEmbedding`
: If this bit (bit 1) is set, the control definition function supports the `kControlMsgSubControlAdded` and `kControlMsgSubControlRemoved` messages.

`kControlSupportsFocus`
: If this bit (bit 2) is set, the control definition function supports the `kControlMsgKeyDown` message. If this bit and the `kControlGetsFocusOnClick` bit are set, the control definition function supports the `kControlMsgFocus` message.

`kControlWantsIdle`
: If this bit (bit 3) is set, the control definition function supports the `kControlMsgIdle` message.

`kControlWantsActivate`
: If this bit (bit 4) is set, the control definition function supports the `kControlMsgActivate` message.

`kControlHandlesTracking`
: If this bit (bit 5) is set, the control definition function supports the `kControlMsgHandleTracking` message.

`kControlSupportsDataAccess`
: If this bit (bit 6) is set, the control definition function supports the `kControlMsgGetData` and `kControlMsgSetData` messages.

`kControlHasSpecialBackground`
: If this bit (bit 7) is set, the control definition function supports the `kControlMsgSetUpBackground` message.

`kControlGetsFocusOnClick`
: If this bit (bit 8) and the `kControlSupportsFocus` bit are set, the control definition function supports the `kControlMsgFocus` message.

`kControlSupportsCalcBestRect`
: If this bit (bit 9) is set, the control definition function supports the `kControlMsgCalcBestRect` message.

`kControlSupportsLiveFeedback`
: If this bit (bit 10) is set, the control definition function supports the `kControlMsgCalcValueFromPos` message.

`kControlHasRadioBehavior`
: If this bit (bit 11) is set, the control definition function supports radio button behavior and can be embedded in a radio group control. This constant is available with Appearance 1.0.1 and later.

Drawing a Ghost Image of the Indicator

If your control definition function supports indicator ghosting, it should return `kControlSupportsGhosting` as one of the feature bits in response to a `kControlMsgGetFeatures` message. If this bit is set and the control indicator is being tracked, the Control Manager calls your control definition function and passes `kControlMsgDrawGhost` in the `message` parameter. A handle to the region where the ghost should be drawn will be passed in the `param` parameter.

Your control definition function should respond by redrawing the control with the ghosted indicator at the specified location and should return 0 as its function result.

Note
The ghost indicator should always be drawn before the actual indicator so that it appears underneath the actual indicator.

Calculating the Optimal Control Rectangle

If your control definition function supports calculating the optimal dimensions of the control rectangle, it should return `kControlSupportsCalcBestRect` as one of the feature bits in response to the `kControlMsgGetFeatures` message. If this bit is set and `GetBestControlRect` (page 152) is called, the Control Manager will call your control definition function and pass `kControlMsgCalcBestRect` in the `message` parameter. The Control Manager passes a pointer to a control size calculation structure in the `param` parameter.

Your control definition function should respond by calculating the width and height of the optimal control rectangle and adjusting the rectangle by setting the `height` and `width` fields of the control size calculation structure to the appropriate values. If your control definition function displays text, it should pass in the offset from the bottom of control to the base of the text in the

baseLine field of the structure. Your control definition function should return the offset value stored in the structure's baseLine field.

The control size calculation structure is a structure of type ControlCalcSizeRec:

```
struct ControlCalcSizeRec {
    SInt16                      height;
    SInt16                      width;
    SInt16                      baseLine;
};
typedef struct ControlCalcSizeRec ControlCalcSizeRec;
typedef ControlCalcSizeRec *ControlCalcSizePtr;
```

Field descriptions

height	The optimal height (in pixels) of the control's bounding rectangle.
width	The optimal width (in pixels) of the control's bounding rectangle.
baseLine	The offset from the bottom of the control to the base of the text. This value is generally negative.

Performing Custom Tracking

If your control definition function supports custom tracking, it should return kControlHandlesTracking as one of the feature bits in response to a kControlMsgGetFeatures message. If this bit is set and a mouse-down event occurs in your control, TrackControl (page 146) or HandleControlClick (page 144) calls your control definition function and passes kControlMsgHandlesTracking in the message parameter. The Control Manager passes a pointer to a control tracking structure in the param parameter. Your control definition function should respond appropriately and return the part code that was hit, or kControlNoPart if the mouse-down event occurred outside the control; see "Control Part Code Constants" (page 99).

The control tracking structure is a structure of type ControlTrackingRec:

```
struct ControlTrackingRec {
    Point                       startPt;
    SInt16                      modifiers;
    ControlActionUPP            action;
```

```
};
typedef struct ControlTrackingRec ControlTrackingRec;
typedef ControlTrackingRec *ControlTrackingPtr;
```

Field descriptions

`startPt` The location of the cursor at the time the mouse button was first pressed, in local coordinates. Your application retrieves this point from the `where` field of the event structure.

`modifiers` The constant in the `modifiers` field of the event structure specifying the state of the modifier keys and the mouse button at the time the event was posted.

`action` A pointer to an action function defining what action your application takes while the user holds down the mouse button. The value of the `actionProc` parameter can be a valid `procPtr`, `nil`, or -1. A value of -1 indicates that the control should either perform auto tracking, or if it is incapable of doing so, do nothing (like `nil`).

Handling Keyboard Focus

If your control definition function can change its keyboard focus, it should set `kControlSupportsFocus` and `kControlGetsFocusOnClick` as feature bits in response to a `kControlMsgGetFeatures` message. If these bits are set and the `AdvanceKeyboardFocus` (page 149), `ReverseKeyboardFocus` (page 150), `ClearKeyboardFocus` (page 151), or `SetKeyboardFocus` (page 147) function is called, the Control Manager calls your control definition function and passes `kControlMsgFocus` in the `message` parameter.

The Control Manager passes one of the control focus part code constants described below or a valid part code in the `param` parameter. Your control definition function should respond by adjusting the focus accordingly.

Your control definition function should return the control focus part code or actual control part that was focused on. Return `kControlFocusNoPart` if your control does not accept focus or has just relinquished it. Return a nonzero part code to indicate that your control received keyboard focus. Your control definition function is responsible for maintaining which part is focused.

```
enum {
    kControlFocusNoPart     = 0,
    kControlFocusNextPart   = -1,
```

```
    kControlFocusPrevPart   = -2
};
typedef SInt16 ControlFocusPart;
```

Constant descriptions

`kControlFocusNoPart`

Your control definition function should relinquish its focus and return `kControlFocusNoPart`. It might respond by deactivating its text edit handle and erasing its focus ring. If the control is at the end of its subparts, it should return `kControlFocusNoPart`. This tells the focusing mechanism to jump to the next control that supports focus.

`kControlFocusNextPart`

Your control definition function should change keyboard focus to its next part, the entire control, or remove keyboard focus from the control, depending upon the circumstances.

For multiple part controls that already had keyboard focus, the next part of the control would receive keyboard focus when `kControlFocusNextPart` was passed in the `param` parameter. For example, a clock control with keyboard focus would change its focus to the left-most element of the control (the month field).

For single-part controls that did not have keyboard focus and are now receiving it, the entire control would receive keyboard focus when `kControlFocusNextPart` was passed in the `param` parameter.

For single-part controls that already had keyboard focus and are now losing it, the entire control would lose keyboard focus.

If you are passed `kControlFocusNextPart` and have run out of parts, return `kControlFocusNoPart` to indicate that the user tabbed past the control.

`kControlFocusPrevPart`

Your control definition function should change keyboard focus to its previous part, the entire control, or remove keyboard focus from the control, depending upon the circumstances.

For multiple part controls that already had keyboard focus, the previous part of the control would receive keyboard

focus when `kControlFocusPrevPart` was passed in the `param` parameter. For example, a clock control with keyboard focus would change its focus to the right-most element of the control (the year field).
For single-part controls that did not have keyboard focus and are now receiving it, the entire control would receive keyboard focus when `kControlFocusNextPart` was passed in the `param` parameter.
For single-part controls that already had keyboard focus and are now losing it, the entire control would lose keyboard focus.
If you are passed `kControlFocusPrevPart` and have run out of parts, return `kControlFocusNoPart` to indicate that the user tabbed past the control.

`<part code>` Your control definition function should focus on the specified part code. Your function can interpret this in any way it wishes.

Handling Keyboard Events

If your control definition function can handle keyboard events, it should return `kControlSupportsFocus`—every control that supports keyboard focus must also be able to handle keyboard events—as one of the feature bits in response to a `kControlMsgGetFeatures` message. If this bit is set, the Control Manager will pass `kControlMsgKeyDown` in the `message` parameter. The Control Manager passes a pointer to a control key down structure in the `param` parameter. Your control definition function should respond by processing the keyboard event as appropriate and return 0 as the function result.

The control key down structure is a structure of type `ControlKeyDownRec`:

```
struct ControlKeyDownRec {
    SInt16                      modifiers;
    SInt16                      keyCode;
    SInt16                      charCode;
};
typedef struct ControlKeyDownRec ControlKeyDownRec;
typedef ControlKeyDownRec *ControlKeyDownPtr;
```

Field descriptions

`modifiers`	The constant in the `modifiers` field of the event structure specifying the state of the modifier keys and the mouse button at the time the event was posted.
`keyCode`	The virtual key code derived from the event structure. This value represents the key pressed or released by the user. It is always the same for a specific physical key on a particular keyboard regardless of which modifier keys were also pressed.
`charCode`	A particular character derived from the event structure. This value depends on the virtual key code, the state of the modifier keys, and the current `'KCHR'` resource.

Performing Idle Processing

If your control definition function can perform idle processing, it should return `kControlWantsIdle` as one of the feature bits in response to a `kControlMsgGetFeatures` message. If this bit is set and `IdleControls` (page 143) is called for the window your control is in, the Control Manager will pass `kControlMsgIdle` in the `message` parameter. Your control definition function should ignore the `param` parameter and respond appropriately. For example, indeterminate progress indicators and asynchronous arrows use idle time to perform their animation.

Your control definition function should return 0 as the function result.

Getting and Setting Control-Specific Data

If your control definition function supports getting and setting control-specific data, it should return `kControlSupportsDataAccess` as one of its features bits in response to the `kControlMsgGetFeatures` message. If this bit is set, the Control Manager will call your control definition function and pass `kControlMsgSetData` in the `message` parameter when `SetControlData` (page 154) is called, and will pass `kControlMsgGetData` in the `message` parameter when `GetControlData` (page 156) and `GetControlDataSize` (page 157) are called. The Control Manager passes a pointer to a control data access structure in the `param` parameter. Your definition function should respond by filling out the structure and returning an operating system status message as the function result.

The control data access structure is a structure of type `ControlDataAccessRec`:

```
struct ControlDataAccessRec{
    ResType             tag;
    ResType             part;
    Size                size;
    Ptr                 dataPtr;
};
typedef struct ControlDataAccessRec ControlDataAccessRec;
typedef ControlDataAccessRec *ControlDataAccessPtr;
```

Field descriptions

`tag` A constant representing a piece of data that is passed in (in response to a `kControlMsgSetData` message) or returned (in response to a `kControlMsgGetData` message); see "Control Data Tag Constants" (page 83) for a description of these constants. The control definition function should return `errDataNotSupported` if the value in the `tag` parameter is unknown or invalid.

`part` The part of the control that this data should be applied to. If the information is not tied to a specific part of the control or the control has no parts, pass 0.

`size` On entry, the size of the buffer pointed to by the `dataPtr` field. In response to a `kControlMsgGetData` message, this field should be adjusted to reflect the actual size of the data that the control is maintaining. If the size of the buffer being passed in is smaller than the actual size of the data, the control definition function should return `errDataSizeMismatch`.

`dataPtr` A pointer to a buffer to read or write the information requested. In response to a `kControlMsgGetData` message, this field could be `nil`, indicating that you wish to return the size of the data in the `size` field.

Handling Activate and Deactivate Events

If your control definition function wants to be informed whenever it is being activated or deactivated, it should return `kControlWantsActivate` as one of the feature bits in response to the `kControlMsgGetFeatures` message. If this bit is set and your control definition function is being activated or deactivated, the Control Manager calls it and passes `kControlMsgActivate` in the `message` parameter. The Control Manager passes a 0 or 1 in the `param` parameter. A value

of 0 indicates that the control is being deactivated; 1 indicates that it is being activated.

Your control definition function should respond by performing any special processing before the user pane becomes activated or deactivated, such as deactivating its `TEHandle` or `ListHandle` if it is about to be deactivated.

Your control definition function should return 0 as the function result.

Setting a Control's Background Color or Pattern

If your control definition function supports embedding and draws its own background, it should return `kControlHasSpecialBackground` as one of the feature bits in response to the `kControlMsgGetFeatures` message. If this bit is set and an embedding hierarchy of controls is being drawn in your control, the Control Manager passes `kControlMsgSetUpBackground` in the `message` parameter of your control definition function. The Control Manager passes a pointer to a filled-in control background structure in the `param` parameter. Your control definition function should respond by setting its background color or pattern to whatever is appropriate given the bit depth and device type passed in. Your control definition function should return 0 as the function result.

The control background structure is a structure of type `ControlBackgroundRec`:

```
struct ControlBackgroundRec {
    SInt16          depth;
    Boolean         colorDevice;
};
typedef struct ControlBackgroundRec ControlBackgroundRec;
typedef ControlBackgroundRec *ControlBackgroundPtr;
```

Field descriptions

`depth` A signed 16-bit integer indicating the bit depth (in pixels) of the current graphics port.

`colorDevice` A Boolean value. If `true`, you are drawing on a color device. If `false`, you are drawing on a monochrome device.

Supporting Live Feedback

If your control definition function supports live feedback while tracking the indicator, it should return `kControlSupportsLiveFeedback` as one of the feature bits in response to the `kControlMsgGetFeatures` message. If this bit is set, the

Control Manager will call your control definition function when it tracks the indicator and pass `kControlMsgCalcValueFromPos` in the `message` parameter. The Control Manager passes a handle to the indicator region being dragged in the `param` parameter.

Your control definition function should respond by calculating its value and drawing the control based on the new indicator region passed in. Your control definition function should not recalculate its indicator position. After the user is done dragging the indicator, your control definition function will be called with a `posCntl` message at which time you can recalculate the position of the indicator. Not recalculating the indicator position each time your control definition function is called creates a smooth dragging experience for the user.

Your control definition function should return 0 as the function result.

Being Informed When Subcontrols Are Added or Removed

If your control definition function wishes to be informed when subcontrols are added or removed, it should return `kControlSupportsEmbedding` as one of the feature bits in response to the `kControlMsgGetFeatures` message. If this bit is set, the Control Manager passes `ControlMsgSubControlAdded` in the `message` parameter immediately after a subcontrol is added, or it passes `kControlMsgSubControlRemoved` just before a subcontrol is removed from your embedder control. A handle to the control being added or removed from the embedding hierarchy is passed in the `param` parameter. Your control definition function should respond appropriately and return 0 as the function result.

Typically, a control definition function only supports this message if it wants to do extra processing in response to changes in its embedded controls. Radio groups use these messages to perform necessary processing for handling embedded controls. For example, if a currently selected radio button is deleted, the group can adjust itself accordingly.

Defining Your Own Action Functions

When your action function is called for a control part, your action function is passed a handle to the control and the control's part code. Your action function should then respond as is appropriate. For an example of such an action function, see `MyActionProc` (page 185). The only exception to this is for indicators that don't support live feedback.

If the mouse-down event occurs in an indicator of a control that does not support live feedback, your action function should take no parameters, because the user may move the cursor outside the indicator while dragging it. For an example of such an action function, see `MyIndicatorActionProc` (page 186).

MyActionProc

CHANGED WITH THE APPEARANCE MANAGER

Defines actions to be performed repeatedly in response to a mouse-down event in a control part.

The Control Manager declares the type for an application-defined action function as follows:

```
typedef pascal void (*ControlActionProcPtr)(
                                        ControlHandle theControl,
                                        ControlPartCode partCode);
```

The Control Manager defines the data type `ControlActionUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlActionUPP;
```

You typically use the `NewControlActionProc` macro like this:

```
ControlActionUPP myActionUPP;
myActionUPP = NewControlActionProc(MyAction);
```

You typically use the `CallControlActionProc` macro like this:

```
CallControlActionProc(MyActionUPP, theControl, partCode);
```

Here's how to declare an action function for a control part if you were to name the function `MyActionProc`:

```
pascal void MyActionProc (
                    ControlHandle theControl,
                    ControlPartCode partCode);
```

`theControl` A handle to the control in which the mouse-down event occurred.

`partCode` A control part code; see "Control Part Code Constants" (page 99). When the cursor is still in the control part where the mouse-down event first occurred, this parameter contains that control's part code. When the user drags the cursor outside the original control part, this parameter contains 0.

DISCUSSION

When a mouse-down event occurs in a control, `HandleControlClick` (page 144) and `TrackControl` (page 146) respond as is appropriate by highlighting the control or dragging the indicator as long as the user holds down the mouse button. You can define other actions to be performed repeatedly during this interval. To do so, define your own action function and point to it in the `actionProc` parameter of the `TrackControl` function or the `inAction` parameter of `HandleControlClick`. This is the only way to specify actions in response to all mouse-down events in a control or indicator.

IMPORTANT

You should use the `MyIndicatorActionProc` function while tracking indicators of controls that don't support live feedback.

SEE ALSO

`SetControlAction` (page 153).

MyIndicatorActionProc

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, you should use `MyActionProc` (page 185) to define actions to be performed in response to a mouse-down event in an indicator of a control that supports live feedback.

You should only use `MyIndicatorActionProc` if the control does not support live feedback.

Defining Your Own Key Filter Function

The key filter function allows for the interception and possible changing of keystrokes destined for a control.

MyControlKeyFilterProc

NEW WITH THE APPEARANCE MANAGER

Controls that support text input (such as editable text and list box controls) can attach a key filter function to filter key strokes and modify them on return.

The Control Manager declares the type for an application-defined key filter function as follows:

```
typedef pascal KeyFilterResult (*ControlKeyFilterProcPtr)(
                                        ControlHandle theControl,
                                        SInt16* keyCode,
                                        SInt16* charCode,
                                        SInt16* modifiers);
```

The Control Manager defines the data type `ControlKeyFilterUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlKeyFilterUPP;
```

You typically use the `NewControlKeyFilterProc` macro like this:

```
NewControlKeyFilterUPP myControlKeyFilterUPP;
myControlKeyFilterUPP = NewControlKeyFilterProc(MyKeyFilter);
```

You typically use the `CallControlKeyFilterProc` macro like this:

```
CallControlKeyFilterProc(myControlKeyFilterUPP, theControl, keyCode,
charCode, modifiers);
```

Here's how to declare a key filter function if you were to name the function `MyControlKeyFilterProc`:

```
pascal ControlKeyFilterResult MyControlKeyFilterProc (
                            ControlHandle theControl,
                            SInt16* keyCode,
                            SInt16* charCode,
                            SInt16* modifiers);
```

`theControl`	A handle to the control in which the mouse-down event occurred.
`keyCode`	The virtual key code derived from the event structure. This value represents the key pressed or released by the user. It is always the same for a specific physical key on a particular keyboard regardless of which modifier keys were also pressed.
`charCode`	A particular character derived from the event structure. This value depends on the virtual key code, the state of the modifier keys, and the current `'KCHR'` resource.
`modifiers`	The constant in the `modifiers` field of the event structure specifying the state of the modifier keys and the mouse button at the time the event was posted.
function result	Returns a value indicating whether or not it allowed or blocked keystrokes; see "Key Filter Result Codes" (page 188).

DISCUSSION

Your key filter function can intercept and change keystrokes destined for a control. Your key filter function can change the keystroke, leave it alone, or block your control definition function from receiving it. For example, an editable text control can use a key filter function to allow only numeric values to be input in its field.

Key Filter Result Codes

Your key filter function returns these constants to specify whether or not a keystroke is filtered or blocked.

```
enum {
    kControlKeyFilterBlockKey   = 0,
    kControlKeyFilterPassKey    = 1
};
typedef SInt16 ControlKeyFilterResult;
```

Constant descriptions

`kControlKeyFilterBlockKey`
The keystroke is blocked and not received by the control.

`kControlKeyFilterPassKey`
The keystroke is filtered and received by the control.

Defining Your Own User Pane Functions

This section describes the application-defined user pane functions that provide you with the ability to create a custom Appearance-compliant control without writing your own control definition function. A **user pane** is a general purpose stub control; it can be used as the root control for a window, as well as providing a way to hook in application-defined functions such as those described below. When Appearance is available, user panes should be used in dialog boxes instead of user items.

Once you have provided a user pane application-defined function, pass the tag constant representing the user pane function you wish to get or set in the `tagName` parameter of `SetControlData` (page 154). For a description of the tag constants, see "Control Data Tag Constants" (page 83). For example, to set a user pane draw function, pass the constant `kControlUserPaneDrawProcTag` of type `ControlUserPaneDrawingUPP` in the `tagName` parameter of `SetControlData` (page 154). The Control Manager then draws the control using a universal procedure pointer to your user pane draw function.

MyUserPaneDrawProc

NEW WITH THE APPEARANCE MANAGER

Draws the content of your user pane control in the rectangle of user pane control.

The Control Manager declares the type for an application-defined user pane draw function as follows:

```
typedef pascal void (*ControlUserPaneDrawProc)(
                                    ControlHandle control,
                                    SInt16 part);
```

The Control Manager defines the data type `ControlUserPaneDrawUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneDrawUPP;
```

You typically use the `NewControlUserPaneDrawProc` macro like this:

```
ControlUserPaneDrawUPP myControlUserPaneDrawUPP;
myControlUserPaneDrawUPP = NewControlUserPaneDrawProc(MyUserPaneDraw);
```

You typically use the `CallControlUserPaneDrawProc` macro like this:

```
CallControlUserPaneDrawProc(myControlUserPaneDrawUPP, control, part);
```

Here's how to declare the function `MyUserPaneDrawProc`:

```
pascal void MyUserPaneDrawProc (
                    ControlHandle control,
                    SInt16 part);
```

`control` A handle to the user pane control in which you wish drawing to occur.

`part` The part code of the control you should draw. If 0, draw the entire control.

DISCUSSION

Once you have created the function `MyUserPaneDrawProc`, pass `kControlUserPaneDrawProcTag` in the `tagName` parameter of `SetControlData` (page 154). The Control Manager will draw the user pane control with a universal procedure pointer to `MyUserPaneDrawProc`.

MyUserPaneHitTestProc

NEW WITH THE APPEARANCE MANAGER

Returns the part code of the control that the point was in when the mouse-down event occurred.

The Control Manager declares the type for an application-defined user pane hit test function as follows:

```
typedef pascal ControlPartCode (*ControlUserPaneHitTestProc) (
                                        ControlHandle control,
                                        Point where);
```

The Control Manager defines the data type `ControlUserPaneHitTestUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneHitTestUPP;
```

You typically use the `NewControlUserPaneHitTestProc` macro like this:

```
ControlUserPaneHitTestUPP myControlUserPaneHitTestUPP;
myControlUserPaneHitTestUPP = NewControlUserPaneHitTestProc
(MyUserPaneHitTest);
```

You typically use the `CallControlUserPaneHitTestProc` macro like this:

```
CallControlUserPaneHitTestProc(myControlUserPaneHitTestUPP, control,
where);
```

Here's how to declare the function `MyUserPaneHitTestProc`:

```
pascal ControlPartCode MyUserPaneHitTestProc (
                    ControlHandle control,
                    Point where);
```

`control`	A handle to the control in which the mouse-down event occurred.
`where`	The point, in a window's local coordinates, where the mouse-down event occurred.
function result	Returns the part code of the control where the mouse-down event occurred. If the point was not over a control, your function should return `kControlNoPart`.

DISCUSSION

Once you have created the function `MyUserPaneHitTestProc`, pass `kControlUserPaneHitTestProcTag` in the `tagName` parameter of `SetControlData` (page 154).

MyUserPaneTrackingProc

NEW WITH THE APPEARANCE MANAGER

Tracks a control while the user holds down the mouse button.

The Control Manager declares the type for an application-defined user pane tracking function as follows:

```
typedef pascal ControlPartCode (*ControlUserPaneTrackingProc)(
                                        ControlHandle control,
                                        Point startPt,
                                        ControlActionUPP actionProc);
```

The Control Manager defines the data type `ControlUserPaneTrackingUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneTrackingUPP;
```

You typically use the `NewControlUserPaneTrackingingProc` macro like this:

```
ControlUserPaneTrackingUPP myControlUserPaneTrackingUPP;
myControlUserPaneTrackingUPP = NewControlUserPaneTrackingProc
(MyUserPaneTracking);
```

You typically use the `CallControlUserPaneTrackingingProc` macro like this:

```
CallControlUserPaneTrackingProc(myControlUserPaneTrackingUPP, control,
startPt, actionProc);
```

Here's how to declare the function `MyUserPaneTrackingProc`:

```
pascal ControlPartCode MyUserPaneTrackingProc (
                    ControlHandle control,
                    Point startPt,
                    ControlActionUPP actionProc);
```

`control` A handle to the control in which the mouse-down event occurred.

`startPt` The location of the cursor at the time the mouse button was first pressed, in local coordinates. Your application retrieves this point from the `where` field of the event structure.

`actionProc` A pointer to an action function defining what action your application takes while the user holds down the mouse button. The value of the `actionProc` parameter can be a valid `procPtr`, `nil`, or -1. A value of -1 indicates that the control should either perform auto tracking, or if it is incapable of doing so, do nothing (like `nil`).

function result Returns the part code of the control part that was tracked. If tracking was unsuccessful, `kControlNoPartCode` is returned.

DISCUSSION

Your `MyUserPaneTrackingProc` function should track the control by repeatedly calling the action function specified in the `actionProc` parameter until the mouse button is released. When the mouse button is released, your function should return the part code of the control part that was tracked.

This function will only get called if you've set the `kControlHandlesTracking` feature bit on creation of the user pane control. Once you have created the function `MyUserPaneTrackingProc`, pass `kControlUserPaneTrackingProcTag` in the `tagName` parameter of `SetControlData` (page 154).

MyUserPaneIdleProc

NEW WITH THE APPEARANCE MANAGER

Performs idle processing.

The Control Manager declares the type for an application-defined user pane idle function as follows:

```
typedef pascal void (*ControlUserPaneIdleProc)(ControlHandle control);
```

The Control Manager defines the data type `ControlUserPaneIdleUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneIdleUPP;
```

You typically use the `NewControlUserPaneIdleProc` macro like this:

```
ControlUserPaneIdleUPP myControlUserPaneIdleUPP;
myControlUserPaneIdleUPP = NewControlUserPaneIdleProc(MyUserPaneIdle);
```

You typically use the `CallControlUserPaneIdleProc` macro like this:

```
CallControlUserPaneIdleProc(myControlUserPaneIdleUPP, control);
```

Here's how to declare the function `MyUserPaneIdleProc`:

```
pascal void MyUserPaneIdleProc (ControlHandle control);
```

`control`	A handle to the control for which you wish to perform idle processing.

DISCUSSION

This function will only get called if you've set the `kControlWantsIdle` feature bit on creation of the user pane control. Once you have created the function `MyUserPaneIdleProc`, pass `kControlUserPaneIdleProcTag` in the `tagName` parameter of `SetControlData` (page 154).

MyUserPaneKeyDownProc

NEW WITH THE APPEARANCE MANAGER

Handles keyboard event processing.

The Control Manager declares the type for an application-defined user pane key down function as follows:

```
typedef pascalControlPartCode(*ControlUserPaneKeyDownProc)(
                                        ControlHandle control
                                        SInt16 keyCode,
                                        SInt16 charCode,
                                        SInt16 modifiers);
```

The Control Manager defines the data type `UserPaneKeyDownUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneKeyDownUPP;
```

You typically use the `NewControlUserPaneKeyDownProc` macro like this:

```
ControlUserPaneKeyDownUPP myControlUserPaneKeyDownUPP;
myControlUserPaneKeyDownUPP = NewControlUserPaneKeyDownProc
(MyUserPaneKeyDown);
```

You typically use the `CallControlUserPaneKeyDownProc` macro like this:

```
CallControlUserPaneKeyDownProc(myControlUserPaneKeyDownUPP, control,
keyCode, charCode, modifiers);
```

Here's how to declare the function `MyUserPaneKeyDownProc`:

```
pascal ControlPartCode MyUserPaneKeyDownProc (
                            ControlHandle control,
                            SInt16 keyCode,
                            SInt16 charCode,
                            SInt16 modifiers);
```

`control`	A handle to the control in which the keyboard event occurred.
`keyCode`	The virtual key code derived from event structure. This value represents the key pressed or released by the user. It is always the same for a specific physical key on a particular keyboard regardless of which modifier keys were also pressed.
`charCode`	A particular character derived from the event structure. This value depends on the virtual key code, the state of the modifier keys, and the current `'KCHR'` resource.
`modifiers`	The constant in the `modifiers` field of the event structure specifying the state of the modifier keys and the mouse button at the time the event was posted.
function result	Returns the part code of the control where the keyboard event occurred. If the keyboard event did not occur in a control, your function should return `kControlNoPart`.

DISCUSSION

Your `MyUserPaneKeyDownProc` function should handle the key pressed or released by the user and return the part code of the control where the keyboard event occurred. This function will only get called if you've set the `kControlSupportsFocus` feature bit on creation of the user pane control. Once you have created the function `MyUserPaneKeyDownProc`, pass `kControlUserPaneKeyDownProcTag` in the `tagName` parameter of `SetControlData` (page 154).

MyUserPaneActivateProc

NEW WITH THE APPEARANCE MANAGER

Handles activate and deactivate event processing.

The Control Manager declares the type for an application-defined user pane activate function as follows:

```
typedef pascal void (*ControlUserPaneActivateProc)(
                                        ControlHandle control,
                                        Boolean activating);
```

The Control Manager defines the data type `UserPaneActivateUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneActivateUPP;
```

You typically use the `NewControlUserPaneActivateProc` macro like this:

```
ControlUserPaneActivateUPP myControlUserPaneActivateUPP;
myControlUserPaneActivateUPP = NewControlUserPaneActivateProc
(MyUserPaneActivate);
```

You typically use the `CallControlUserPaneActivateProc` macro like this:

```
CallControlUserPaneActivateProc(myControlUserPaneActivateUPP, control,
activating);
```

Here's how to declare the function `MyUserPaneActivateProc`:

```
pascal void MyUserPaneActivateProc (
                    ControlHandle control
                    Boolean activating);
```

`control` A handle to the control in which the activate event occurred.

`activating` A Boolean value indicating whether or not the control is being activated. If `true`, the control is being activated. If `false`, the control is being deactivated.

DISCUSSION

Your `MyUserPaneActivateProc` function should perform any special processing before the user pane becomes activated or deactivated. For example, it should deactivate its `TEHandle` or `ListHandle` if the user pane is about to be deactivated.

This function will only get called if you've set the `kControlWantsActivate` feature bit on creation of the user pane control. Once you have created the function `MyUserPaneActivateProc`, pass `kControlUserPaneActivateProcTag` in the `tagName` parameter of `SetControlData` (page 154).

MyUserPaneFocusProc

NEW WITH THE APPEARANCE MANAGER

Handles keyboard focus.

The Control Manager declares the type for an application-defined user pane focus function as follows:

```
typedef pascal ControlPartCode (*ControlUserPaneFocusProc)(
                                        ControlHandle control,
                                        ControlFocusPart action);
```

The Control Manager defines the data type `ControlUserPaneFocusUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneFocusUPP;
```

You typically use the `NewControlUserPaneFocusProc` macro like this:

```
ControlUserPaneFocusUPP myControlUserPaneFocusUPP;
myControlUserPaneFocusUPP = NewControlUserPaneFocusProc
(MyUsePaneFocus);
```

You typically use the `CallControlUserPaneFocusProc` macro like this:

```
CallControlUserPaneFocusProc(myControlUserPaneFocusUPP, control, action);
```

Here's how to declare the function `MyUserPaneFocusProc`:

```
pascal ControlPartCode MyUserPaneFocusProc (
                    ControlHandle control
                    ControlFocusPart action);
```

`control` A handle to the control that is to adjust its focus.

`action`	The part code of the user pane to receive keyboard focus; see "Handling Keyboard Focus" (page 178).
function result	Returns the part of the user pane actually focused. `kControlFocusNoPart` is returned if the user pane has lost the focus or cannot be focused.

DISCUSSION

Your `MyUserPaneFocusProc` function is called in response to a change in keyboard focus. It should respond by changing keyboard focus based on the part code passed in the `action` parameter.

This function will only get called if you've set the `kControlSupportsFocus` feature bit on creation of the user pane control. Once you have created the function `MyUserPaneFocusProc`, pass `kControlUserPaneFocusProcTag` in the `tagName` parameter of `SetControlData` (page 154).

MyUserPaneBackgroundProc

NEW WITH THE APPEARANCE MANAGER

Sets the background color or pattern (only for user panes that support embedding).

The Control Manager declares the type for an application-defined user pane focus function as follows:

```
typedef pascal (*ControlUserPaneBackgroundProcPtr)(
                                        ControlHandle control,
                                        ControlBackgroundPtr info);
```

The Control Manager defines the data type `ControlUserPaneBackgroundUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr ControlUserPaneBackgroundUPP;
```

You typically use the `NewControlUserPaneBackgroundProc` macro like this:

```
ControlUserPaneBackgroundUPP myControlUserPaneBackgroundUPP;
myControlUserPaneBackgroundUPP = NewControlUserPaneBackgroundProc
(MyUsePaneBackground);
```

You typically use the `CallControlUserPaneBackgroundProc` macro like this:

```
CallControlUserPaneBackgroundProc(myControlUserPaneBackgroundUPP,
control, info);
```

Here's how to declare the function `MyUserPaneBackgroundProc`:

```
pascal void MyUserPaneBackgroundProc (
                    ControlHandle control
                    ControlBackgroundPtr info);
```

`control`	A handle to the control for which the background color or pattern is to be set.
`info`	A pointer to information such as the depth and type of the drawing device.

DISCUSSION

Your `MyUserPaneFocusProc` function should set the user pane background color or pattern to whatever is appropriate given the bit depth and device type passed in.

This function will only get called if you've set the `kControlHasSpecialBackground` and `kControlSupportsEmbedding` feature bits on creation of the user pane control. Once you have created the function `MyUserPaneBackgroundProc`, pass `kControlUserPaneBackgroundProcTag` in the `tagName` parameter of `SetControlData` (page 154).

Window Manager Reference

Contents

This chapter describes the Window Manager types, constants, resources, and functions that are affected by Mac OS 8 or the Appearance Manager.

- "Window Manager Types and Constants" (page 203) lists Window Manager types and constants, including structures. Result codes are included at the end of this section.
- "Window Manager Resources" (page 220) describes the window (`'WIND'`) resource, the window color table (`'wctb'`) resource, and the window definition function (`'WDEF'`) resource.
- "Window Manager Functions" (page 223) describes both Window Manager functions and application-defined functions.

Window Manager Types and Constants

Window Definition IDs

CHANGED WITH THE APPEARANCE MANAGER

A window definition ID is supplied to the window resource (page 220) or to a window-creation function to specify which window definition function to use in creating the window. A variation code may also be used to describe variations of the same basic window.

The window definition ID is an integer that contains the resource ID of the window definition function in its upper 12 bits and a variation code in its lower 4 bits. For a given resource ID and variation code, the window definition ID is derived as follows:

window definition ID = (16 x resource ID) + variation code

If you wish to create a custom window, you can write your own window definition function. For an example, see `MyWindowDefProc` (page 231).

IMPORTANT

The window definition IDs for dialog boxes and utility (floating) windows pertain to the appearances of these windows only, not their behaviors. For example, if you want a utility window to have the proper behavior, that is, float, your application must provide for it.

When mapping is enabled, standard pre-Appearance window definition function IDs will be mapped to their Appearance-compliant equivalents. For a discussion of how to enable mapping, see "Introduction to the Appearance Manager" (page 19). Table 3-1 (page 205) lists the standard pre-Appearance and Appearance-compliant window definition ID constants.

Table 3-1 Pre-Appearance and Appearance-compliant window definition IDs

Pre-Appearance window	Appearance-compliant window	Description
dBoxProc	kWindowModalDialogProc	Modal dialog box
None	kWindowAlertProc	Modal alert box
movableDBoxProc	kWindowMovableModalDialogProc	Movable modal dialog box
None	kWindowMovableAlertProc	Movable modal alert box
plainDBox	kWindowPlainDialogProc	Modeless dialog box
altDBoxProc	kWindowShadowDialogProc	Modeless dialog box with shadow
noGrowDocProc	kWindowDocumentProc	Movable window with no size box
documentProc	kWindowGrowDocumentProc	Movable window with size box
zoomNoGrow	kWindowFullZoomDocumentProc	Movable window with full zoom box and no size box
zoomDocProc	kWindowFullZoomGrowDocumentProc	Movable window with full zoom box and size box
None	kWindowVertZoomDocumentProc	Movable window with vertical zoom box and no size box
None	kWindowVertZoomGrowDocumentProc	Movable window with vertical zoom box and size box
None	kWindowHorizZoomDocumentProc	Movable window with horizontal zoom box and no size box
None	kWindowHorizZoomGrowDocumentProc	Movable window with horizontal zoom box and size box

Pre-Appearance window	Appearance-compliant window	Description
`rDocProc`	None (use `rDocProc`)	Round-cornered window
`floatProc`	`kWindowFloatProc`	Utility window with no size box or zoom box
`floatGrowProc`	`kWindowFloatGrowProc`	Utility window with size box
`floatZoomProc`	`kWindowFloatFullZoomProc`	Utility window with full zoom box
`floatZoomGrowProc`	`kWindowFloatFullZoomGrowProc`	Utility window with full zoom box and size box
None	`kWindowFloatVertZoomProc`	Utility window with vertical zoom box
None	`kWindowFloatVertZoomGrowProc`	Utility window with vertical zoom box and size box
None	`kWindowFloatHorizZoomProc`	Utility window with horizontal zoom box
None	`kWindowFloatHorizZoomGrowProc`	Utility window with horizontal zoom box and size box
`floatSideProc`	`kWindowFloatSideProc`	Utility window with side title bar and no size box or zoom box
`floatSideGrowProc`	`kWindowFloatSideGrowProc`	Utility window with side title bar and size box
`floatSideZoomProc`	`kWindowFloatSideFullZoomProc`	Utility window with side title bar and full zoom box

Pre-Appearance window	Appearance-compliant window	Description
floatSideZoomGrowProc	kWindowFloatSideFullZoomGrowProc	Utility window with side title bar, size box, and full zoom box
None	kWindowFloatSideVertZoomProc	Utility window with side title bar and vertical zoom box
None	kWindowFloatSideVertZoomGrowProc	Utility window with side title bar, vertical zoom box, and size box
None	kWindowFloatSideHorizZoomProc	Utility window with side title bar and horizontal zoom box
None	kWindowFloatSideHorizZoomGrowProc	Utility window with side title bar, horizontal zoom box, and size box

```
enum {
    documentProc                    = 0,
    dBoxProc                        = 1,
    plainDBox                       = 2,
    altDBoxProc                     = 3,
    noGrowDocProc                   = 4,
    movableDBoxProc                 = 5,
    zoomDocProc                     = 8,
    zoomNoGrow                      = 12,
    rDocProc                        = 16,
    kWindowDocumentProc             = 1024,
    kWindowGrowDocumentProc         = 1025,
    kWindowVertZoomDocumentProc     = 1026,
    kWindowVertZoomGrowDocumentProc = 1027,
    kWindowHorizZoomDocumentProc    = 1028,
    kWindowHorizZoomGrowDocumentProc = 1029,
    kWindowFullZoomDocumentProc     = 1030,
    kWindowFullZoomGrowDocumentProc = 1031,
```

```
    kWindowPlainDialogProc              = 1040,
    kWindowShadowDialogProc             = 1041,
    kWindowModalDialogProc              = 1042,
    kWindowMovableModalDialogProc       = 1043,
    kWindowAlertProc                    = 1044,
    kWindowMovableAlertProc             = 1045,
    kWindowFloatProc                    = 1057,
    kWindowFloatGrowProc                = 1059,
    kWindowFloatVertZoomProc            = 1061,
    kWindowFloatVertZoomGrowProc        = 1063,
    kWindowFloatHorizZoomProc           = 1065,
    kWindowFloatHorizZoomGrowProc       = 1067,
    kWindowFloatFullZoomProc            = 1069,
    kWindowFloatFullZoomGrowProc        = 1071,
    kWindowFloatSideProc                = 1073,
    kWindowFloatSideGrowProc            = 1075,
    kWindowFloatSideVertZoomProc        = 1077,
    kWindowFloatSideVertZoomGrowProc    = 1079,
    kWindowFloatSideHorizZoomProc       = 1081,
    kWindowFloatSideHorizZoomGrowProc   = 1083,
    kWindowFloatSideFullZoomProc        = 1085,
    kWindowFloatSideFullZoomGrowProc    = 1087,
    floatProc                           = 1985,
    floatGrowProc                       = 1987,
    floatZoomProc                       = 1989,
    floatZoomGrowProc                   = 1991,
    floatSideProc                       = 1993,
    floatSideGrowProc                   = 1995,
    floatSideZoomProc                   = 1997,
    floatSideZoomGrowProc               = 1999
};
```

Constant descriptions

`documentProc`	Pre-Appearance document window (movable window with size box).
`dBoxProc`	Pre-Appearance modal dialog box.
`plainDBox`	Pre-Appearance modeless dialog box.
`altDBoxProc`	Pre-Appearance modeless dialog box with shadow.
`noGrowDocProc`	Pre-Appearance movable window with no size box or zoom box.

`movableDBoxProc` Pre-Appearance movable modal dialog box.

`zoomDocProc` Pre-Appearance movable window with size box and full zoom box.

`zoomNoGrow` Pre-Appearance window with full zoom box and no size box.

`rDocProc` Pre-Appearance rounded-corner window. You can control the diameter of curvature of a rounded-corner window (window type `rDocProc`) by adding one of these integers to the `rDocProc` constant:

Window definition ID	Diameters of curvature
`rDocProc`	16, 16
`rDocProc + 2`	4, 4
`rDocProc + 4`	6, 6
`rDocProc + 6`	10, 10

`kWindowDocumentProc`
Appearance-compliant movable window with no size box or zoom box.

`kWindowGrowDocumentProc`
Appearance-compliant standard document window (movable window with size box).

`kWindowVertZoomDocumentProc`
Appearance-compliant window with vertical zoom box and no size box.

`kWindowVertZoomGrowDocumentProc`
Appearance-compliant window with vertical zoom box and size box.

`kWindowHorizZoomDocumentProc`
Appearance-compliant window with horizontal zoom box and no size box.

`kWindowHorizZoomGrowDocumentProc`
Appearance-compliant window with horizontal zoom box and size box.

`kWindowFullZoomDocumentProc`
Appearance-compliant window with full zoom box and no size box.

`kWindowFullZoomGrowDocumentProc`
: Appearance-compliant window with full zoom box and size box.

`kWindowPlainDialogProc`
: Appearance-compliant modeless dialog box.

`kWindowShadowDialogProc`
: Appearance-compliant modeless dialog box with shadow.

`kWindowModalDialogProc`
: Appearance-compliant modal dialog box.

`kWindowMovableModalDialogProc`
: Appearance-compliant movable modal dialog box.

`kWindowAlertProc`
: Appearance-compliant alert box.

`kWindowMovableAlertProc`
: Appearance-compliant movable alert box.

`kWindowFloatProc`
: Appearance-compliant utility (floating) window with no size box or zoom box.

`kWindowFloatGrowProc`
: Appearance-compliant utility (floating) window with a size box.

`kWindowFloatVertZoomProc`
: Appearance-compliant utility (floating) window with a vertical zoom box.

`kWindowFloatVertZoomGrowProc`
: Appearance-compliant utility (floating) window with a vertical zoom box and size box.

`kWindowFloatHorizZoomProc`
: Appearance-compliant utility (floating) window with a horizontal zoom box.

`kWindowFloatHorizZoomGrowProc`
: Appearance-compliant utility (floating) window with a horizontal zoom box and size box.

`kWindowFloatFullZoomProc`
: Appearance-compliant utility (floating) window with full zoom box.

`kWindowFloatFullZoomGrowProc`
: Appearance-compliant utility (floating) window with full zoom box and size box.

`kWindowFloatSideProc`
: Appearance-compliant utility (floating) window with side title bar.

`kWindowFloatSideGrowProc`
: Appearance-compliant utility (floating) window with side title bar and size box.

`kWindowFloatSideVertZoomProc`
: Appearance-compliant utility (floating) window with side title bar and vertical zoom box.

`kWindowFloatSideVertZoomGrowProc`
: Appearance-compliant utility (floating) window with side title bar, vertical zoom box, and size box.

`kWindowFloatSideHorizZoomProc`
: Appearance-compliant utility (floating) window with side title bar and horizontal zoom box.

`kWindowFloatSideHorizZoomGrowProc`
: Appearance-compliant utility (floating) window with side title bar, horizontal zoom box, and size box.

`kWindowFloatSideFullZoomProc`
: Appearance-compliant utility (floating) window with side title bar and full zoom box.

`kWindowFloatSideFullZoomGrowProc`
: Appearance-compliant utility (floating) window with side title bar, full zoom box, and size box.

`floatProc` Pre-Appearance utility (floating) window with no size box or zoom box.

`floatGrowProc` Pre-Appearance utility (floating) window with size box.

`floatZoomProc` Pre-Appearance utility (floating) window with zoom box.

`floatZoomGrowProc` Pre-Appearance utility (floating) window with size box and zoom box.

`floatSideProc` Pre-Appearance utility (floating) window with side title bar and no size or zoom box.

`floatSideGrowProc` Pre-Appearance utility (floating) window with side title bar and size box.

`floatSideZoomProc` Pre-Appearance utility (floating) window with side title bar and zoom box.

`floatSideZoomGrowProc`
Pre-Appearance utility (floating) window with side title bar, size box, and zoom box.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Only the following window definition ID constants are supported:

```
documentProc                = 0,
dBoxProc                    = 1,
plainDBox                   = 2,
altDBoxProc                 = 3,
noGrowDocProc               = 4,
movableDBoxProc             = 5,
zoomDocProc                 = 8,
zoomNoGrow                  = 12,
rDocProc                    = 16,
floatProc                   = 1985,
floatGrowProc               = 1987,
floatZoomProc               = 1989,
floatZoomGrowProc           = 1991,
floatSideProc               = 1993,
floatSideGrowProc           = 1995,
floatSideZoomProc           = 1997,
floatSideZoomGrowProc       = 1999
```

Window Resource IDs

CHANGED WITH THE APPEARANCE MANAGER

You can use one of these constants to create a window definition ID. The standard Appearance-compliant resource ID constants `kWindowDocumentDefProcResID`, `kWindowUtilityDefProcResID`, and `kWindowUtilitySideTitleDefProcResID` have collapse boxes.

Note
Resource IDs 0 through 127 are reserved for use by the system.

```
enum {
    kStandardWindowDefinition           = 0,
    kRoundWindowDefinition              = 1,
    kWindowDocumentDefProcResID         = 64,
    kWindowDialogDefProcResID           = 65,
    kWindowUtilityDefProcResID          = 66,
    kWindowUtilitySideTitleDefProcResID = 67,
    kFloatingWindowDefinition           = 124
};
```

Constant descriptions

`kStandardWindowDefinition`

Defines pre-Appearance standard document windows and dialog boxes. When mapping is enabled, this resource ID is mapped to `kWindowDocumentDefProcResID` or `kWindowDialogDefProcResID`. When mapped to `kWindowDocumentDefProcResID`, this produces an Appearance-compliant standard document window with no size box and no vertical or horizontal zoom box. When mapped to `kWindowDialogDefProcResID`, this produces an Appearance-compliant dialog box with no size box and a 3-pixel space between the dialog box's content and structure region.

`kRoundWindowDefinition`

Defines pre-Appearance standard desk-accessory style windows. This resource ID is not mapped to any Appearance-compliant resource ID when mapping is enabled.

`kWindowDocumentDefProcResID`

Defines Appearance-compliant standard document windows with a size box. Standard document windows created with this resource ID can use variation codes to create windows with vertical and horizontal zoom boxes.

`kWindowDialogDefProcResID`

Defines Appearance-compliant dialog and alert boxes. Modal and movable modal dialog boxes created with this resource ID are displayed with no space between their content and structure region. Alert boxes created with this resource ID are displayed with a red-tinged border.

`kWindowUtilityDefProcResID`
: Defines Appearance-compliant utility (floating) windows with a top title bar and a size box.

`kWindowUtilitySideTitleDefProcResID`
: Defines Appearance-compliant utility (floating) windows with a side title bar and a size box.

`kFloatingWindowDefinition`
: Defines pre-Appearance utility (floating) windows. When mapping is enabled, this resource ID is mapped to `kWindowUtilityDefProcResID` or `kWindowUtilitySideTitleDefProcResID`. When mapped to `kWindowUtilityDefProcResID`, this produces an Appearance-compliant utility window with no size box until `DrawGrowIcon` (page 227) is called. When mapped to `kWindowUtilitySideTitleDefProcResID`, it produces an Appearance-compliant utility window with a side title bar and no size box until `DrawGrowIcon` (page 227) is called.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Only the following window resource ID constants are supported:

```
kStandardWindowDefinition          = 0,
kRoundWindowDefinition             = 1,
kFloatingWindowDefinition          = 124
```

Window Definition Function Variation Codes

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, you should use the Appearance-compliant window definition function ID constants described in "Window Definition IDs" (page 203), rather than variation codes.

Window Region Constants

NEW WITH THE APPEARANCE MANAGER

You can pass constants of type `WindowRegionCode` in the `inRegionCode` parameter of `GetWindowRegion` (page 226) to obtain a handle to a specific window region.

Figure 3-1 (page 216) illustrates the location of these regions in a window.

```
enum {
    kWindowTitleBarRgn      = 0,
    kWindowTitleTextRgn     = 1,
    kWindowCloseBoxRgn      = 2,
    kWindowZoomBoxRgn       = 3,
    kWindowDragRgn          = 5,
    kWindowGrowRgn          = 6,
    kWindowCollapseBoxRgn   = 7,
    kWindowStructureRgn     = 32,
    kWindowContentRgn       = 33
};
typedef UInt16 WindowRegionCode;
```

Constant descriptions

`kWindowTitleBarRgn` The entire area occupied by a window's title bar, including the title text region.

`kWindowTitleTextRgn` That portion of a window's title bar that is occupied by the name of the window.

`kWindowCloseBoxRgn` The area occupied by a window's close box.

`kWindowZoomBoxRgn` The area occupied by a window's zoom box.

`kWindowDragRgn` The draggable area of the window frame, including the title bar and window outline, but excluding the close box, zoom box, and collapse box.

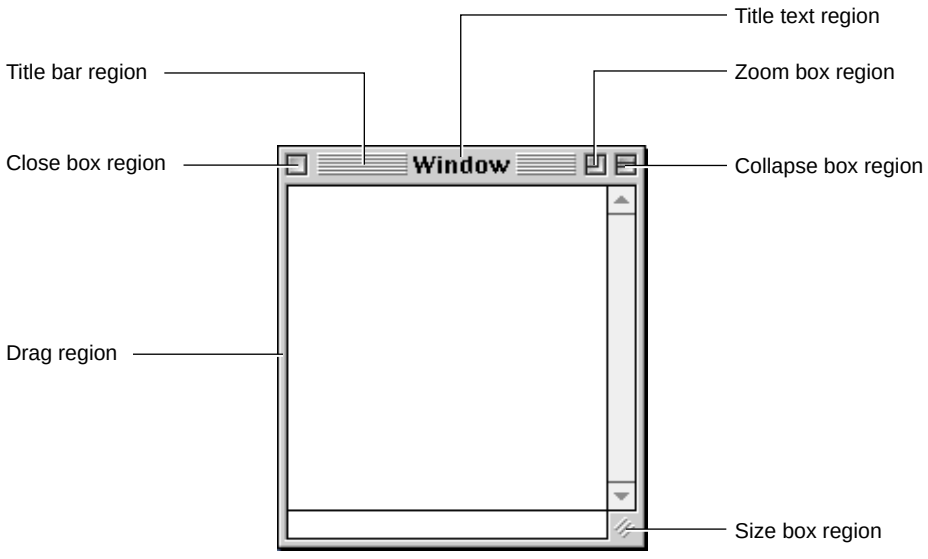
`kWindowGrowRgn` The area occupied by a window's size box.

`kWindowCollapseBoxRgn` The area occupied by a window's collapse box.

`kWindowStructureRgn` The entire area occupied by a window, including the frame and content region; the window may be partially off-screen but its structure region does not change.

`kWindowContentRgn` Window's content region (the part of a window in which your application displays the contents of the window or dialog, including the size box and any controls).

Figure 3-1 Window regions



Part Identifier Constants

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard windows, all the fields of the window color table structure are ignored except the part identifier constant `wContentColor` in the `value` field of the `ColorSpec` structure, which produces the background color for the window's content region.

If you are creating your own custom windows, the window color table structure and all its part identifier constants can still be used.

FindWindow Result Code Constants

CHANGED WITH THE APPEARANCE MANAGER

When your application receives a mouse-down event, you typically call `FindWindow` (page 224). `FindWindow` returns an integer that specifies the location, in global coordinates, of the cursor at the time the user pressed the mouse button.

```
enum {
    inDesk          = 0,
    inMenuBar       = 1,
    inSysWindow     = 2,
    inContent       = 3,
    inDrag          = 4,
    inGrow          = 5,
    inGoAway        = 6,
    inZoomIn        = 7,
    inZoomOut       = 8,
    inCollapseBox   = 11
};
```

Constant descriptions

`inDesk` The cursor is in the desktop region, not in the menu bar, a driver window, or any window that belongs to your application. When `FindWindow` returns `inDesk`, your application doesn't need to do anything.

`inMenuBar` The user has pressed the mouse button while the cursor is in the menu bar. When `FindWindow` returns `inMenuBar`, your application typically adjusts its menus and then calls the Menu Manager function `MenuSelect` to let the user choose menu items.

`inSysWindow` The user has pressed the mouse button while the cursor is in a window belonging to a driver in your application's partition. If `FindWindow` returns `inSysWindow`, your application typically calls the function `SystemClick`.

`inContent` The user has pressed the mouse button while the cursor is in the content area (excluding the size box in an active window) of one of your application's windows. When

	FindWindow returns inContent, your application determines how to handle clicks in the content region.
inDrag	The user has pressed the mouse button while the cursor is in the drag region of a window. When FindWindow returns inDrag, your application typically calls DragWindow to let the user drag the window to a new location.
narrow	The user has pressed the mouse button while the cursor is in an active window's size box. When FindWindow returns inGrow, your application typically calls GrowWindow .
inGoAway	The user has pressed the mouse button while the cursor is in an active window's close box. When FindWindow returns inGoAway, your application typically calls TrackGoAway to track mouse activity while the button is down and then calls its own function for closing a window if the user releases the button while the cursor is in the close box.
inZoomIn	The user has pressed the mouse button while the cursor is in the zoom box of an active window that is currently in the standard state. When FindWindow returns inZoomIn, your application typically calls TrackBox to track mouse activity while the button is down and then calls its own function for zooming a window if the user releases the button while the cursor is in the zoom box.
inZoomOut	The user has pressed the mouse button while the cursor is in the zoom box of an active window that is currently in the user state. When FindWindow returns inZoomOut, your application typically calls the function TrackBox to track mouse activity while the button is down. Your application then calls its own function for zooming a window if the user releases the button while the cursor is in the zoom box.
inCollapseBox	The user has pressed the mouse button while the cursor is in an active window's collapse box. When FindWindow returns inCollapseBox, your application typically does nothing, because the system will collapse your window for you.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

The inCollapseBox constant will not be returned by FindWindow.

The Window Structure

NOT RECOMMENDED WITH MAC OS 8

In Mac OS 8, you should use the color window structure instead of the window structure, since Color QuickDraw is always available with Mac OS 8.

The Window State Data Structure

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, you should not extend the window state data structure. Instead use the `refCon` field of the color window structure or extend the window record structure.

The Window Color Table Structure

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard windows, all information in the window color table structure is ignored except the part identifier constant `wContentColor` in the `value` field of the `ColorSpec` structure. This constant produces the background color for the window's content region. If your are creating your own custom window definition functions, the window color table structure can still be used.

The Auxiliary Window Structure

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard windows, most of the fields of the auxiliary window structure are ignored. In the future, standard system windows may not have entries in the auxiliary window list. If you are creating your own window definition function, the auxiliary window structure can still be used.

Result Codes

The most common result codes that can be returned by Window Manager functions are listed below.

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough memory
`resNotFound`	–192	Unable to read resource

Window Manager Resources

The Window Resource

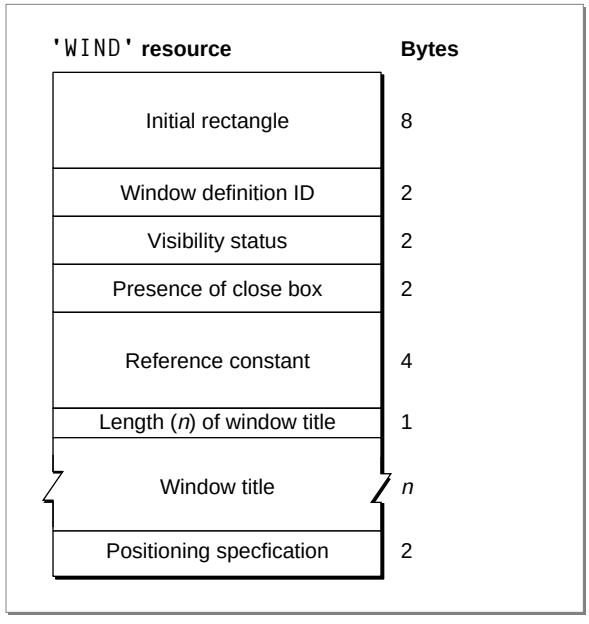
CHANGED WITH THE APPEARANCE MANAGER

You typically define a window (`'WIND'`) resource for each type of window that your application creates. Use `GetNewWindow` (page 223) or `GetNewCWindow` to create a window based on a `'WIND'` resource. Both functions create a new window structure and fill it in according to the values specified in the `'WIND'` resource.

Note
Window resources must have resource ID numbers greater than 127.

Figure 3-2 illustrates a compiled `'WIND'` resource.

Figure 3-2 Structure of a compiled window `('WIND')` resource



A compiled version of a window resource contains the following elements:

- The upper-left and lower-right corners, in global coordinates, of a rectangle that defines the initial size and placement of the window's content region. Your application can change this rectangle before displaying the window, either programmatically or through an optional positioning code described later in this section.
- The window definition ID. The window definition ID is an integer that contains the resource ID of the window definition function in its upper 12 bits and a variation code in its lower 4 bits; see "Window Definition IDs" (page 203) for a discussion of standard pre-Appearance and Appearance-compliant window definition IDs.
- A specification that determines whether the window will be visible or invisible. This characteristic controls only whether the Window Manager displays the window, not necessarily whether the window can be seen on the screen. (A visible window entirely covered by other windows, for example, is "visible" even though the user cannot see it.) You typically create

a new window in an invisible state, build the content area of the window, and then display the completed window.

- A specification that determines whether or not the window will have a close box. The `'WDEF'` draws the close box when it draws the window frame. The window type specified in the second field determines whether a window can support a close box; this field determines whether the close box is present.
- A reference constant, which your application can use for whatever data it needs to store. When it builds a new window structure, the Window Manager stores, in the `refCon` field, whatever value you specify in the fifth element of the window resource. You can also put a place-holder value here and then set the `refCon` field programmatically with the `SetWRefCon` function.
- A pascal string that specifies the window title.
- A positioning specification that overrides the window position established by the rectangle in the first field. The existence of this field is optional. The positioning constants are convenient when the user is creating new documents or when you are handling your own dialog boxes and alert boxes. When you are creating a new window to display a previously saved document, however, you should display the new window in the same rectangle as that in which it was previously displayed.

The Window Color Table Resource

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard windows, all information in the window color table `('wctb')` resource is ignored except the part identifier constant `wContentColor` in the `value` field of the `ColorSpec` structure. This constant produces the background color for the window's content region.

If your are creating your own custom window definition functions, the window color table resource can still be used.

The Window Definition Function Resource

CHANGED WITH THE APPEARANCE MANAGER

The window definition (`'WDEF'`) resource is the executable code for your window definition function. Provide as the resource data the compiled or assembled code of your window definition procedure. The entry point of your procedure must be at the beginning of the resource data. Under Appearance, there are two new values that the Window Manager can pass in the `message` parameter to your `'WDEF'`, `kWindowMsgGetFeatures` and `kWindowMsgGetRegion`.

See `MyWindowDefProc` (page 231) for more information about creating a window definition function.

▲ **WARNING**
All resources of type `'WDEF'` should be nonpurgeable, to ensure that the resource is always loaded. If the resource is not available when the Window Manager needs to load it, a crash will occur.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

The messages `kWindowMsgGetFeatures` and `kWindowMsgGetRegion` will not be sent to your window definition function.

Window Manager Functions

Creating and Closing Windows

GetNewWindow

NOT RECOMMENDED WITH MAC OS 8

Creates a new monochrome window from a window resource. The `GetNewWindow` function was originally implemented before the advent of Color

QuickDraw. In Mac OS 8, you should call the Color QuickDraw function `GetNewCWindow` (see the "Window Manager" chapter in *Inside Macintosh: Macintosh Toolbox Essentials*) instead of `GetNewWindow` to programmatically create a window, because Color QuickDraw is always available in Mac OS 8.

NewWindow

NOT RECOMMENDED WITH MAC OS 8

Creates a new monochrome window with the characteristics specified by a list of parameters. The `NewWindow` function was originally implemented before the advent of Color QuickDraw. In Mac OS 8, you should call the Color QuickDraw function `NewCWindow` (see the "Window Manager" chapter in *Inside Macintosh: Macintosh Toolbox Essentials*) instead of `NewWindow` to programmatically create a window, because Color QuickDraw is always available in Mac OS 8.

Retrieving Window Information

FindWindow

CHANGED WITH THE APPEARANCE MANAGER

Maps the location of the cursor to a part of the screen or a region of a window when your application receives a mouse-down event.

```
pascal short FindWindow (
                    Point thePoint,
                    WindowPtr *theWindow);
```

`thePoint` The point, in global coordinates, where the mouse-down event occurred. Your application retrieves this information from the `where` field of the event structure.

`theWindow` A pointer to the window in which the mouse-down event occurred. `FindWindow` produces `nil` if the mouse-down event occurred outside a window.

function result Returns a short integer that specifies where the cursor was when the user pressed the mouse button; see "FindWindow Result Code Constants" (page 217).

DISCUSSION

You typically call the function `FindWindow` whenever you receive a mouse-down event. The `FindWindow` function helps you dispatch the event by reporting whether the cursor was in the menu bar or in a window when the mouse button was pressed. If the cursor was in a window, the function will produce both a pointer to the window and a constant that identifies the region of the window in which the event occurred. If Appearance is available, `FindWindow` may return the `inCollapseBox` constant as one of the possible window regions.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

The `inCollapseBox` constant will not be returned by `FindWindow`.

GetWindowFeatures

NEW WITH THE APPEARANCE MANAGER

Gets the features a window supports.

```
pascal OSStatus GetWindowFeatures (
                    WindowPtr inWindow,
                    UInt32* outFeatures);
```

`inWindow` On input, a pointer to the window whose features you wish to determine.

`outFeatures` On output, a pointer to a bit field specifying the features the window supports; see "Reporting Window Features" (page 239).

function result A result code; see "Result Codes" (page 220).

DISCUSSION

The `GetWindowFeatures` function produces a window definition function's features in response to a `kWindowMsgGetFeatures` message. For a list of the

features a window might support, see "Reporting Window Features" (page 239).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

GetWindowRegion

NEW WITH THE APPEARANCE MANAGER

Gets a handle to a specific window region.

```
pascal OSStatus GetWindowRegion (
                    WindowPtr inWindow,
                    WindowRegionCode inRegionCode,
                    RgnHandle ioWinRgn);
```

`inWindow`	On input, a pointer to the window whose region you wish to receive a handle to.
`inRegionCode`	Pass a constant representing the window region whose handle you wish to obtain; see "Window Region Constants" (page 215).
`ioWinRgn`	On input, a handle to a region created by your application. On output, a handle to the specified window region.
function result	A result code; see "Result Codes" (page 220).

DISCUSSION

The `GetWindowRegion` function produces a handle to a window definition function's window region in response to a `kWindowMsgGetRegion` message. The visibility of the window is unimportant for `GetWindowRegion` to work correctly.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

Displaying Windows

DrawGrowIcon

CHANGED WITH THE APPEARANCE MANAGER

Draws a window's size box.

```
pascal void DrawGrowIcon (WindowPtr theWindow);
```

`theWindow` On input, a pointer to the window structure.

DISCUSSION

When you adopt Appearance directly, you never need to call the `DrawGrowIcon` function to get a size box in your window. You can still use `DrawGrowIcon` to draw the delimiting scroll bar lines, if you wish.

If you are going through the mapping layer, however, you do need to call `DrawGrowIcon`, but only once, because under Appearance, once `DrawGrowIcon` is called, the size box is merged into the window's frame.

The `DrawGrowIcon` function doesn't erase the scroll bar areas. If you use `DrawGrowIcon` to draw the size box and scroll bar outline, therefore, you should erase those areas yourself when the window size changes, even if the window doesn't contain scroll bars.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

The `DrawGrowIcon` function draws a window's size box or, if the window can't be sized, whatever other image is appropriate. You call `DrawGrowIcon` when drawing the content region of a window that contains a size box.

The exact appearance and location of the image depend on the window type and the window's active or inactive state. `DrawGrowIcon` automatically checks the window's type and state and draws the appropriate image.

In an active document window, `DrawGrowIcon` draws the grow image in the size box in the lower-right corner of the window's graphics port rectangle, along with the lines delimiting the size box and scroll bar areas. To draw the size box

but not the scroll bar outline, set the `clipRgn` field in the window's graphics port to be a 15-by-15 pixel rectangle in the lower-right corner of the window.

In an inactive document window, `DrawGrowIcon` draws the lines delimiting the size box and scroll bar areas and erases the size box.

Collapsing Windows

CollapseWindow

NEW WITH THE APPEARANCE MANAGER

Collapses or expands a window to its title bar .

```
pascal OSStatus CollapseWindow (
                    WindowPtr inWindow,
                    Boolean inCollapseIt);
```

`inWindow`	On input, a pointer to the window that you want to collapse.
`inCollapseIt`	A Boolean value indicating whether the window should be collapsed or expanded.
function result	A result code; see "Result Codes" (page 220).

DISCUSSION

The `CollapseWindow` function tells the Window Manager to collapse or expand a window depending upon the value passed in the `inCollapseIt` parameter. Only window definition functions that return the feature bit `kWindowCanCollapse` in response to a `kWindowGetFeatures` message support this function; see `GetWindowFeatures` (page 225).

SEE ALSO

`FindWindow` (page 224).

"Appearance Manager Gestalt Selector Constants" (page 21).

CollapseAllWindows

NEW WITH THE APPEARANCE MANAGER

Collapses or expands all windows that are collapsable in an application.

```
pascal OSStatus CollapseAllWindows (Boolean inCollapseEm);
```

`inCollapseEm` A Boolean value. Set to `true` to collapse all windows in the application; if `false`, expands all windows in the application.

function result A result code; see "Result Codes" (page 220).

DISCUSSION

Only window definition functions that return the feature bit `kWindowCanCollapse` in response to a `kWindowGetFeatures` message support the `CollapseAllWindows` function; see `GetWindowFeatures` (page 225).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

IsWindowCollapsed

NEW WITH THE APPEARANCE MANAGER

Determines a window's collapse state.

```
pascal Boolean IsWindowCollapsed (WindowPtr inWindow);
```

`inWindow` On input, a pointer to the window whose collapse state you are determining.

function result A Boolean value. If `true`, the window is collapsed. If `false`, the window is expanded.

DISCUSSION

Only window definition functions that return the feature bit `kWindowCanCollapse` in response to a `kWindowGetFeatures` message support this

function; see `GetWindowFeatures` (page 225). Your window definition function should call `IsWindowCollapsed` to determine whether or not a window is collapsed, so you can modify its structure and content regions as appropriate. Typically, a window's content region is empty in a collapsed state.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

Setting and Getting Window Characteristics

GetAuxWin

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard windows, most of the fields of the auxiliary window structure are ignored and the `GetAuxWin` function is not recommended. In the future, standard system windows may not have entries in the auxiliary window list. If you are creating your own window definition function, `GetAuxWin` can still be used.

SetWinColor

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and your application uses standard windows, the `SetWinColor` function is not recommended. `SetWinColor` sets a window color table structure which is now mostly ignored. Only the part identifier constant `wContentColor` in the `value` field of the `ColorSpec` structure is used. This constant produces the background color for the window's content region. Under Appearance, instead of the `SetWinColor` function, the window definition function determines what colors it will use.

If your are creating your own custom window definition functions, `SetWinColor` can still be used.

Defining Your Own Window Definition Function

A window definition function determines how a menu generally looks and behaves. Various Window Manager functions call a window definition function whenever they need to perform a window-dependent action, such as drawing the window on the screen.

The Window Manager calls the Resource Manager to access your window definition function with the given resource ID; see "Window Definition IDs" (page 203) for a description of how window definition IDs are derived from resource IDs and variation codes. You can define your own window variation codes so that you can use one `'WDEF'` resource to handle several variations of the same general window.

The Resource Manager reads your window definition function into memory and returns a handle to it. The Window Manager stores this handle in the `windowDefProc` field of the window structure. Later, when it needs to perform an action on the window, the Window Manager calls the window definition function and passes it the variation code as a parameter.

MyWindowDefProc

CHANGED WITH THE APPEARANCE MANAGER

Your window definition function is responsible for

- drawing the window frame
- reporting the region where mouse-down events occur
- calculating the window's structure region and content region
- drawing the size box
- resizing the window frame when the user drags the size box
- reporting the window's features or the location of a specific window region
- performing any customized initialization or disposal tasks

If you wish to define new, nonstandard windows for your application, you must write a window definition function and store it in a resource file as a resource of type `'WDEF'`.

The Window Manager declares the type for an application-defined window definition function as follows:

```
typedef pascal long (*WindowDefProcPtr)(
                    short varCode,
                    WindowPtr theWindow,
                    short message,
                    long param);
```

The Window Manager defines the data type `WindowDefUPP` to identify the universal procedure pointer for this application-defined function:

```
typedef UniversalProcPtr WindowDefUPP;
```

You typically use the `NewWindowDefProc` macro like this:

```
WindowDefUPP myWindowDefUPP;
myWindowDefUPP = NewWindowDefProc(MyWindow);
```

You typically use the `CallWindowDefProc` macro like this:

```
CallWindowDefProc(myWindowDefUPP, varCode, theWindow, message, param);
```

Here's how to declare the function `MyWindowDefProc`:

```
pascal long MyWindowDef(
                    short varCode,
                    WindowPtr theWindow,
                    short message,
                    long param);
```

`varCode` The window's variation code.

`theWindow` A pointer to the window's window structure.

`message` A value indicating the task to be performed. The `message` parameter contains one of the values defined in "Messages" (page 233). The subsections that follow explain each of these tasks in detail.

`param` Data associated with the task specified by the `message` parameter. If the task requires no data, this parameter is ignored.

function result Your window definition function should perform whatever task is specified by the `message` parameter and return a function result, if appropriate. If the task performed requires no result code, return 0.

Messages

The Window Manager passes a value defined by one of these constants in the `message` parameter of your window definition function to specify the action your function must perform. Other messages are reserved for internal use by the system.

```
enum {
    wDraw                   = 0,
    wHit                    = 1,
    wCalcRgns               = 2,
    wNew                    = 3,
    wDispose                = 4,
    wGrow                   = 5,
    wDrawGIcon              = 6,
    kWindowMsgGetFeatures   = 7,
    kWindowMsgGetRegion     = 8
};
```

Constant descriptions

`wDraw` Draw the window's frame.

`wHit` Report the location of a mouse-down event.

`wCalcRgns` Calculate the structure region and the content region.

`wNew` Perform additional initialization.

`wDispose` Perform additional disposal.

`wGrow` Draw the dotted outline of the window that you see during a resizing operation.

`wDrawGIcon` Draw the outlines for the size box and the scroll bar.

`kWindowMsgGetFeatures` Report the window's features.

`kWindowMsgGetRegion` Report the location of a specific window region.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Only the following messages will be sent to your window definition function:

```
enum {
    wDraw                   = 0,
    wHit                    = 1,
    wCalcRgns               = 2,
    wNew                    = 3,
    wDispose                = 4,
    wGrow                   = 5,
    wDrawGIcon              = 6
} ;
```

Drawing the Window Frame

When the Window Manager passes `wDraw` in the `message` parameter, your window definition function should respond by drawing the window frame in the current graphics port (which is the Window Manager port). The window part code to be drawn will be passed in the `param` parameter of your window definition function.

Your window definition function should perform the following steps:

- Change the current port from the `WMgrPort` to the `WMgrCPort` to allow the system to draw in the full range of RGB colors.
- Update the pen attributes, text attributes, and `bkPat` fields in the `WMgrCPort` to the values of the corresponding fields in the `WMgrPort`. The Window Manager automatically transfers the `vis` and `clip` regions.

Note
The parallelism of the `WMgrPort` and the `WMgrCPort` is maintained only by the window definition functions. All window definition functions that draw in the `WMgrPort` should follow the steps listed above even if the changed fields do not affect their operation.

You must make certain checks to determine exactly how to draw the frame. If the value of the `visible` field in the window structure is `false`, you should do nothing; otherwise, you should examine the `param` parameter and the status flags in the window structure:

- If the value of `param` is 0, draw the entire window frame (including the size box, if your window definition function incorporates the size box into the frame).
- If the value of `param` is 0 and the `hilited` field in the window structure is `true`, highlight the frame to show that the window is active.
  - If the value of the `goAwayFlag` field in the window structure is also `true`, draw a close box in the window frame.
  - If the value of the `spareFlag` field in the window structure is also `true`, draw a zoom box in the window frame.
- If the value of the `param` parameter is `wInGoAway`, add highlighting to, or remove it from, the window's close box.
- If the value of the `param` parameter is `wInZoom`, add highlighting to, or remove it from, the window's zoom box.
- If the value of the `param` parameter is `wInCollapseBox`, add highlighting to, or remove it from, the window's collapse box.

You need to maintain your own state flag to determine whether the close, zoom, or collapse box is to be drawn as highlighted. Typically, you clear this state flag whenever you draw the entire frame, and you set it before drawing whenever your application is called to draw just the close, zoom, or collapse box. If the flag is set, you draw the box in a highlighted state.

The window frame typically, but not necessarily, includes the window's title, which should be displayed in the system font and system font size. The Window Manager port is already set to use the system font and system font size.

Note
Nothing drawn outside the window's structure region will be visible.

Your window definition function should return 0 as the function result for this message.

Reporting the Region of a Mouse-Down Event

When the Window Manager passes `wHit` in the `message` parameter, your window definition function should respond by reporting the region of the specified mouse-down event. The mouse location (in global coordinates) of the window frame will be passed into the `param` parameter of your window

definition function. The vertical coordinate is in the high-order word of the parameter, and the horizontal coordinate is in the low-order word.

In response to the `wHit` message, your window definition function should return one of the following constants:

```
enum {
    wNoHit          = 0,
    wInContent      = 1,
    wInDrag         = 2,
    wInGrow         = 3,
    wInGoAway       = 4,
    wInZoomIn       = 5,
    wInZoomOut      = 6,
    wInCollapseBox  = 9
};
```

Constant descriptions

`wNoHit` The mouse-down event did not occur in the content region or the drag region of any active or inactive window or in the close, size, zoom, or collapse box of an active window. The return value `wNoHit` might also mean that the point isn't in the window. The standard window definition functions, for example, return `wNoHit` if the point is in the window frame but not in the title bar.

`wInContent` The mouse-down event occurred in the content region of an active or inactive window (with the exception of the size box).

`wInDrag` The mouse-down event occurred in the drag region of an active or inactive window.

`wInGrow` The mouse-down occurred in the size box of an active window.

`wInGoAway` The mouse-down event occurred in the close box of an active window.

`wInZoomIn` The mouse-down event occurred in the zoom box of an active window that is currently in the standard state.

`wInZoomOut` The mouse-down event occurred in the zoom box of an active window that is currently in the user state.

`wInCollapseBox`
: The mouse-down event occurred in the collapse box of an active window.

Return the constants `wInGrow`, `wInGoAway`, `wInZoomIn`, `wInZoomOut`, and `wInCollapseBox` only if the window is active—by convention, the size box, close box, zoom box, and collapse box aren't drawn if the window is inactive. In an inactive document window, for example, a mouse-down event in the part of the title bar that would contain the close box if the window were active is reported as `wInDrag`.

Calculating Regions

When the Window Manager passes `wCalcRgns` in the `message` parameter, your window definition function should respond by calculating the window's structure and content regions based on the current graphics port's port rectangle. These regions, whose handles are in the `strucRgn` and `contRgn` fields of the window structure, are in global coordinates. The Window Manager requests this operation only if the window is visible. The mouse location (in global coordinates) of the window frame will be passed into the `param` parameter of your window definition function.

Your window definition function should call `IsWindowCollapsed` (page 229) to determine its collapse state. Then your window definition function can modify its structure and content regions as appropriate. Typically, a window's content region is empty in a collapsed state.

▲ **WARNING**
When you calculate regions for your own type of window, do not alter the clip region or the visible region of the Window Manager port. The Window Manager and QuickDraw take care of this for you. Altering the Window Manager port's clip region or visible region may damage other windows.

Your window definition function should return 0 as the function result for this message.

Performing Additional Window Initialization

When the Window Manager passes `wNew` in the `message` parameter, your window definition function should respond by performing any initialization that it may require. If the content region has an unusual shape, for example,

you might allocate memory for the region and store the region handle in the `dataHandle` field of the window structure. The initialization function for a standard document window creates the `wStateData` structure for storing zooming data.

Your window definition function should ignore the `param` parameter and return 0 as the function result for this message.

Performing Additional Window Disposal Actions

When the Window Manager passes `wDispose` in the `message` parameter, your window definition function should respond by performing any additional tasks necessary for disposing of a window. You might, for example, release memory that was allocated by the initialization function. The dispose function for a standard document window disposes of the `wStateData` structure.

Your window definition function should ignore the `param` parameter and return 0 as the function result for this message.

Drawing the Window's Grow Image

When the Window Manager passes `wGrow` in the `message` parameter, your window definition function should respond to being resized by drawing a dotted outline of the window in the current graphics port in the pen pattern and mode. (The pen pattern and mode are set up—as `gray` and `notPatXor`—to conform to Appearance-compliant human interface guidelines.)

A rectangle (in global coordinates) whose upper-left corner is aligned with the port rectangle of the window's graphics port is passed into the `param` parameter of your window definition function. Your grow image should be sized appropriately for the specified rectangle. As the user drags the mouse, the Window Manager sends repeated `wGrow` messages, so that you can change your grow image to match the changing mouse location.

`DrawGrowIcon` (page 227) draws a dotted (gray) outline of the window and also the lines delimiting the title bar, size box, and scroll bar areas.

Your window definition function should return 0 as the function result for this message.

Drawing the Size Box

When the Window Manager passes `wDrawGIcon` in the `message` parameter, your window definition function should respond by drawing the size box in the

content region if the window is active. If the window is inactive, your window definition function should draw whatever is appropriate to show that the window cannot currently be sized. Your window definition function may also draw scroll bar delimiter lines. Your window definition function should ignore the `param` parameter.

If the size box is located in the window frame, draw the size box in response to a `wDraw` message, not a `wDrawGIcon` message.

Your window definition function should return 0 as the function result for this message.

Reporting Window Features

When the Window Manager passes `kWindowMsgGetFeatures` in the `message` parameter, your window definition function should respond by setting the `param` parameter to reflect the features that your window supports. The value passed back in the `param` parameter should be comprised of one or more of the following values:

```
enum{
    kWindowCanGrow              = (1 << 0),
    kWindowCanZoom              = (1 << 1),
    kWindowCanCollapse          = (1 << 2),
    kWindowIsModal              = (1 << 3),
    kWindowCanGetWindowRegion   = (1 << 4),
    kWindowIsAlert              = (1 << 5),
    kWindowHasTitleBar          = (1 << 6),
};
```

Constant descriptions

`kWindowCanGrow` If this bit (bit 0) is set, the window has a grow box (may not be visible).

`kWindowCanZoom` If this bit (bit 1) is set, the window has a zoom box (may not be visible).

`kWindowCanCollapse` If this bit (bit 2) is set, the window has a collapse box.

`kWindowIsModal` If this bit (bit 3) is set, the window should behave as modal.

`kWindowCanGetWindowRegion` If this bit (bit 4) is set, the window supports a call to `GetWindowRegion` (page 226).

kWindowIsAlert	If this bit (bit 5) is set, the window is an alert box (may be movable or not). When this constant is added to kWindowIsModal, the user should be able to switch out of the application and move the alert box.
kWindowHasTitleBar	If this bit (bit 6) is set, the window has a title bar.

Your window definition function should return 1 as the function result for this message.

Returning the Location of Window Regions

When the Window Manager passes kWindowMsgGetRegion in the message parameter, your window definition function should respond by returning the location (in global coordinates) of the specified window region. A pointer to a window region structure will be passed in the param parameter.

The window region structure is a structure of type GetWindowRegionRec.

```
struct GetWindowRegionRec {
    RgnHandle           winRgn;
    WindowRegionCode    regionCode;
};
typedef struct GetWindowRegionRec GetWindowRegionRec;
typedef GetWindowRegionRec *GetWindowRegionPtr;
```

Field descriptions

winRgn	A handle to a window region based on the value specified in the regionCode field. Modify this region.
regionCode	A value representing a given window region; see "Window Region Constants" (page 215).

Your window definition function should return an operating system status (OSStatus) message as the function result for this message. The result code errWindowRgnInvalid indicates that the window region passed in was not valid.

Dialog Manager Reference

Contents

This chapter describes the Dialog Manager types, constants, resources, and functions that are affected by Mac OS 8 or the Appearance Manager.

- "Dialog Manager Types and Constants" (page 243) lists Dialog Manager types and constants, including structures. Result codes are included at the end of this section.
- "Dialog Manager Resources" (page 249) describes the new Appearance-compliant resources for the Dialog Manager and identifies the other Dialog Manager resources that are unchanged or no longer recommended.
- "Dialog Manager Functions" (page 261) describes Dialog Manager functions you can call to invoke alerts, create and dispose of dialog boxes, manipulate items in alert and dialog boxes, and handle events in alert and dialog boxes.

Dialog Manager Types and Constants

Alert Type Constants

NEW WITH THE APPEARANCE MANAGER

You can pass constants of type `AlertType` in the `inAlertType` parameter of `StandardAlert` (page 261) to specify the type of alert box you wish to create.

```
enum {
    kAlertStopAlert     = 0,
    kAlertNoteAlert     = 1,
    kAlertCautionAlert  = 2,
    kAlertPlainAlert    = 3
};
typedef SInt16 AlertType;
```

Constant descriptions

`kAlertStopAlert` Stop alert box.

`kAlertNoteAlert` Note alert box.

`kAlertCautionAlert` Caution alert box.

`kAlertPlainAlert` Alert box with no icon.

Dialog Feature Flag Constants

NEW WITH THE APPEARANCE MANAGER

You can set the following bits in the dialog flags field of the extended dialog resource (page 249) or pass them in the `inFlags` parameter of `NewFeaturesDialog` (page 270) to specify the dialog box's Appearance-compliant features.

```
enum {
    kDialogFlagsUseThemeBackground      = (1 << 0),
    kDialogFlagsUseControlHierarchy     = (1 << 1),
    kDialogFlagsHandleMovableModal      = (1 << 2),
    kDialogFlagsUseThemeControls        = (1 << 3)
};
```

Constant descriptions

`kDialogFlagsUseThemeBackground`
If this bit (bit 0) is set, the Dialog Manager sets the dialog box's background color or pattern.

`kDialogFlagsUseControlHierarchy`
If this bit (bit 1) is set, the Dialog Manager creates a root control in the dialog box and establishes an embedding hierarchy. Any dialog items become controls once the embedding hierarchy is established.

`kDialogFlagsHandleMovableModal`
If this bit (bit 2) is set, and the dialog box is a movable modal (specify the `kWindowMovableModalDialogProc` window definition ID), the Dialog Manager handles movable modal behavior such as dragging a dialog box by its title bar or switching out of the application by clicking in another one.

`kDialogFlagsUseThemeControls`
If this bit (bit 3) is set, the Dialog Manager creates Appearance-compliant controls in the dialog box directly. Otherwise, push buttons, checkboxes, and radio buttons will be displayed in their pre-Appearance forms when systemwide Appearance is off.

Alert Feature Flag Constants

NEW WITH THE APPEARANCE MANAGER

You can set the following bits in the alert flags field of the extended alert resource (page 253) to specify the alert box's Appearance-compliant features.

```
enum {
    kAlertFlagsUseThemeBackground           = (1 << 0),
    kAlertFlagsUseControlHierarchy          = (1 << 1),
    kAlertFlagsAlertIsMovable               = (1 << 2),
    kAlertFlagsUseThemeControls             = (1 << 3)
};
```

`kAlertFlagsUseThemeBackground`
If this bit (bit 0) is set, the Dialog Manager sets the alert box's background color or pattern.

`kAlertFlagsUseControlHierarchy`
If this bit (bit 1) is set, the Dialog Manager creates a root control in the alert box and establishes an embedding hierarchy. Any alert items become controls once the embedding hierarchy is established.

`kAlertFlagsAlertIsMovable`
If this bit (bit 2) is set, the alert box is movable modal. The Dialog Manager handles movable modal behavior such as dragging the alert box by its title bar or switching out of the application by clicking in another one.

`kAlertFlagsUseThemeControls`
If this bit (bit 3) is set, the Dialog Manager creates Appearance-compliant controls in your alert box. Otherwise, push buttons, checkboxes, and radio buttons will be displayed in their pre-Appearance forms when systemwide Appearance is off.

The Standard Alert Structure

NEW WITH THE APPEARANCE MANAGER

A standard alert structure of type `AlertStdAlertParamRec` can be used when you call the function `StandardAlert` (page 261) to customize the alert box.

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

Field descriptions

`movable` A Boolean value indicating whether or not the alert box is movable.

`helpButton` A Boolean value indicating whether or not the alert includes a Help button.

`filterProc` If the value in the `movable` field is `true` (alert is movable), then specify in this parameter a universal procedure pointer to an application-defined filter function that responds to events not handled by `ModalDialog` (page 281). If you do, all events will get routed to your application-defined event filter function for handling, even when your alert box window is in the background. If you set this parameter to `nil`, the Dialog Manager uses the standard event filter function.

`defaultText` Text for button in OK position; see "Alert Default Text Constants" (page 248). The button automatically sizes and positions itself in the alert box. To specify that the default button names should be used, pass `-1`. To indicate that no button should be displayed, pass `nil`.

`cancelText`	Text for button in Cancel position; see "Alert Default Text Constants" (page 248). The button automatically sizes and positions itself in the alert box. To specify that the default button names should be used, pass `-1`. To indicate that no button should be displayed, pass `nil`.
`otherText`	Text for button in leftmost position; see "Alert Default Text Constants" (page 248). The button automatically sizes and positions itself in the alert box. To specify that the default button names should be used, pass `-1`. To indicate that no button should be displayed, pass `nil`.
`defaultButton`	Specifies which button acts as the default button; see "Alert Button Constants" (page 247).
`cancelButton`	Specifies which button acts as the Cancel button (can be 0); see "Alert Button Constants" (page 247).
`position`	The alert box position, as defined by a window positioning constant; see *Macintosh Toolbox Essentials* (page 4-126) for a discussion of these constants. In this structure, the constant `kWindowDefaultPosition` is equivalent to the constant `kWindowAlertPositionParentWindowScreen`.

Alert Button Constants

NEW WITH THE APPEARANCE MANAGER

You can use these constants in the `defaultButton` and `cancelButton` fields in the standard alert structure (page 246) to specify which buttons act as the default and Cancel buttons in the standard alert structure. These constants are also returned in the `itemHit` parameter of `StandardAlert` (page 261).

```
enum {
    kAlertStdAlertOKButton      = 1,
    kAlertStdAlertCancelButton  = 2,
    kAlertStdAlertOtherButton   = 3,
    kAlertStdAlertHelpButton    = 4
};
```

Constant descriptions

`kAlertStdAlertOKButton`
The OK button. The default text for this button is "OK".

`kAlertStdAlertCancelButton`
: The Cancel button (optional). The default text for this button is “Cancel”.

`kAlertStdAlertOtherButton`
: A third button (optional). The default text for this button is “Don’t Save”.

`kAlertStdAlertHelpButton`
: The Help button (optional).

Alert Default Text Constants

NEW WITH THE APPEARANCE MANAGER

You can use these constants in the `defaultText`, `cancelText`, and `otherText` fields of the standard alert structure (page 246) to specify the default text for the OK, Cancel, and Don’t Save buttons.

```
enum {
    kAlertDefaultOKText     = -1,
    kAlertDefaultCancelText = -1,
    kAlertDefaultOtherText  = -1
};
```

Constant descriptions

`kAlertDefaultOKText`
: The default text for the default (right) button is “OK” on an English system. The text will vary depending upon the localization of the user’s system. Use this constant in the `defaultText` field of the standard alert structure (page 246).

`kAlertDefaultCancelText`
: The default text for the Cancel (middle) button is “Cancel” on an English system. The text will vary depending upon the localization of your system. Use this constant in the `cancelText` field of the standard alert structure (page 246).

`kAlertDefaultOtherText`
: The default text for the third (leftmost) button is “Don’t Save” for an English system. The text will vary depending upon the localization of the user’s system. Use this

constant in the `otherText` field of the standard alert structure.

Result Codes

The most common result codes that can be returned by Dialog Manager functions are listed below.

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough memory
`resNotFound`	–192	Unable to read resource
`hmHelpManagerNotInited`	–855	Help Manager not set up

Dialog Manager Resources

The Dialog Resource

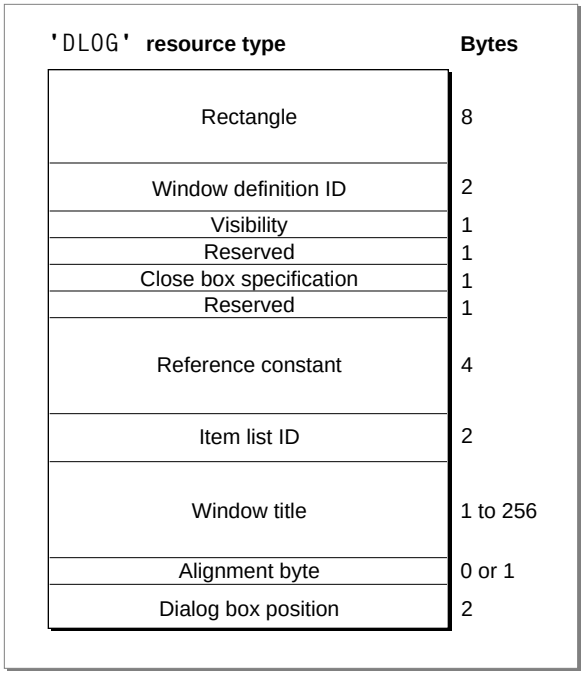
CHANGED WITH THE APPEARANCE MANAGER

A dialog resource is a resource of type `'DLOG'`. All dialog resources can be marked purgeable, and they must have resource ID numbers greater than 127.

You can use an extended dialog resource (page 252) with the same resource ID as your dialog resource to provide additional features for your dialog box, including movable modal behavior, Appearance-compliant backgrounds and controls, and embedding hierarchies. To specify the items in an dialog box, you must provide an item list resource. Use the `GetNewDialog` (page 268) function to create the dialog box defined in the dialog resource and extended dialog resource.

Figure 4-1 shows the format of a compiled dialog resource.

Figure 4-1 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. See "Window Definition IDs" (page 203) for descriptions of the ID constants that can be used in this field to specify Appearance-compliant modal, movable modal, or modeless dialog boxes.
- 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 (page 268). 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 function `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 here that represents a dialog box type, in order to distinguish between a number of similar dialog boxes. As this information is stored in a window structure within the dialog structure, you can use the Window Manager function `SetWRefCon` at any time to change this value in the dialog structure for a dialog box, and you can use the Window Manager function `GetWRefCon` to determine its current value.
- Item list resource ID. This ID specifies the item list resource and the dialog font table resource that will be used with this dialog box.
- Window title. A string displayed in the dialog box's title bar only when the dialog box is modeless or movable modal.
- Alignment byte. This is an extra byte added if necessary to make the previous Pascal string end on a word boundary.
- Dialog box position. A constant specifying the position of the dialog box on the screen; see the discussion of window positioning constants in *Macintosh Toolbox Essentials* (page 4-126). If your application positions dialog boxes on its own, you shouldn't use these constants, because they may cause conflicts with the Dialog Manager.
  - If 0x0000 appears here (as specified by the `kWindowDefaultPosition` 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 `kWindowAlertPositionParentWindow` 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.
  - If 0x300A appears here (as specified by the `kWindowAlertPositionMainScreen` constant in the Rez input file), the Dialog Manager centers the dialog box near the top of the main screen.
  - If 0x700A appears here (as specified in the Rez input file by the `kWindowAlertPositionParentWindowScreen` constant), the Dialog Manager positions the dialog box on the screen where the user is currently working.

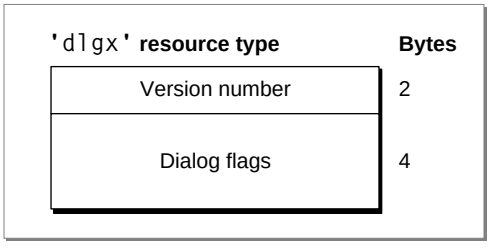
The Extended Dialog Resource

NEW WITH THE APPEARANCE MANAGER

Use an extended dialog resource with the same resource ID as your dialog resource to provide additional features for your dialog box, including movable modal behavior, theme-compliant backgrounds and controls, and embedding hierarchies. An extended dialog resource is a resource of type `'dlgx'`. All extended dialog resources can be marked purgeable, and they must have the same resource ID and be located in the same file as their corresponding dialog resource. Use the function `GetNewDialog` (page 268) to create the dialog box defined in the dialog resource and extended dialog resource.

Figure 4-2 shows the format of a compiled extended dialog resource.

Figure 4-2 Structure of a compiled extended dialog `('dlgx')` resource



The compiled version of an extended dialog resource contains the following elements:

- Version number. An integer specifying the version of the format of the resource.
- Dialog flags. Constants that specify the dialog box's Appearance features; see "Dialog Feature Flag Constants" (page 244).

The Extended Alert Resource

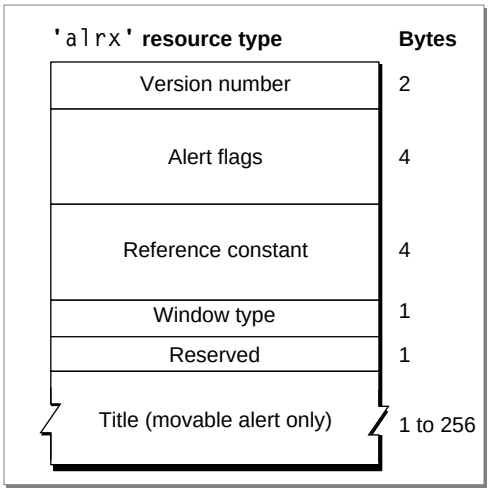
NEW WITH THE APPEARANCE MANAGER

You can use an extended alert resource with the same resource ID as your alert resource to provide additional features for your alert box, including movable modal behavior, Appearance-compliant backgrounds and controls, and embedding hierarchies. The resource also gives you the option of creating a title for movable alert boxes.

Note
Alert titles are only available with Appearance version 1.0.1 and later.

An extended alert resource is a resource of type `'alrx'`. All extended alert resources can be marked purgeable, and they must have the same resource ID and resource file as their corresponding alert resource. Figure 4-3 shows the structure of a compiled extended alert resource.

Figure 4-3 Structure of a compiled extended alert `('alrx')` resource



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

- Version number. An integer specifying the version of the format of the resource.
- Alert flags. Constants specifying the alert box's Appearance features; see "Alert Feature Flag Constants" (page 245).
- Reference constant. This contains any value that an application wishes to store here. For example, an application can store a number here that represents an alert box type, in order to distinguish between a number of similar alert boxes. As this information is stored in a window structure within the dialog structure, you can use `GetWRefCon` to determine this value.
- Window type. If this Boolean is set to 1 (`true`), the Dialog Manager specifies an Appearance-compliant window definition ID constant directly when drawing the alert box window.
- Reserved. Set to 0.
- Window title. A string representing the title of a movable alert box.

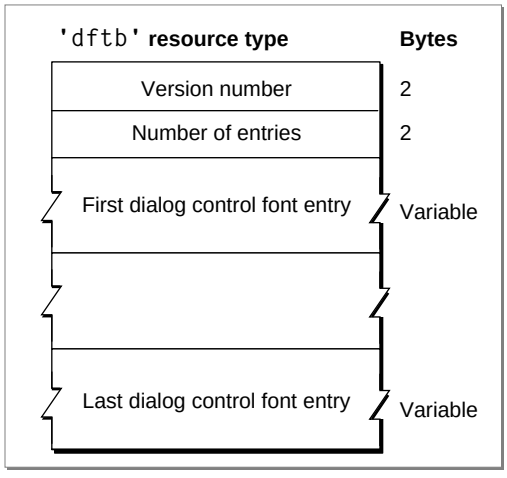
The Dialog Control Font Table Resource

NEW WITH THE APPEARANCE MANAGER

Your application can specify the initial font settings for all controls in a dialog box or alert box by creating a dialog control font table resource of type `'dftb'` with the same resource ID as the item list resource (`'DITL'`). When an embedding hierarchy is established in a dialog box, the dialog control font table resource should be used instead of the item color table (`'ictb'`) resource, since edit and static text dialog items become controls in an embedding hierarchy.

The control font style information in the dialog control font table resource is automatically read in (along with the `'DITL'`) by the Dialog Manager. When the `'dftb'` resource is read in, the control font styles are set, and the resource is marked purgeable. Figure 4-4 shows the format of a compiled dialog control font table resource.

Figure 4-4 Structure of a compiled dialog control font table `('dftb')` resource

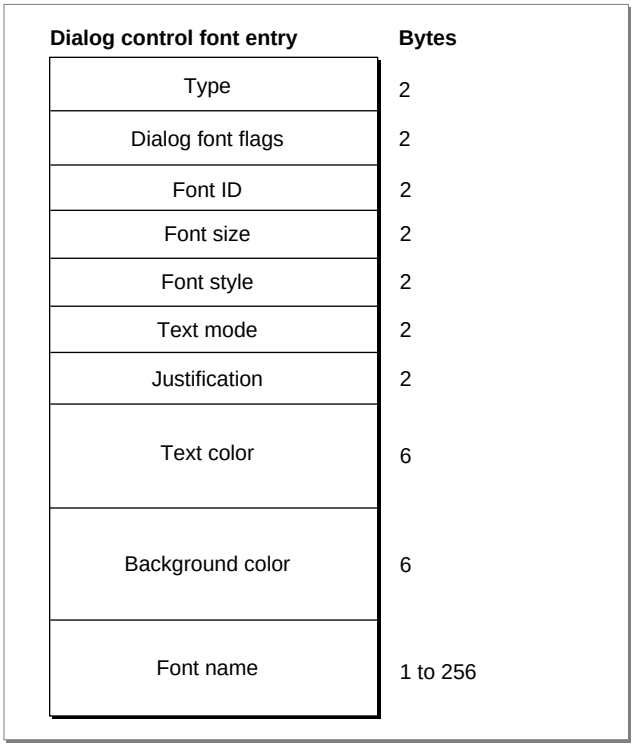


A compiled version of a `'dftb'` resource contains the following elements:

- Version number. An integer specifying the version of the format of the resource.
- Number of entries. An integer that specifies the number of entries in the resource. Each entry is a dialog control font structure.
- Dialog control font entries. A series of dialog control font structures, each of which consist of type, dialog font flags, the font ID, font size, font style, text mode, justification, text color, background color, and font name.

Figure 4-5 shows the format of a compiled dialog control font entry in a `'dftb'` resource.

Figure 4-5 Structure of dialog control font entry in a `'dftb'` resource



Each entry in a `'dftb'` resource corresponds to a dialog item and contains the following elements:

- Type. An integer that specifies whether there is font information for the dialog or alert item in the `'DITL'`. If you specify a value of 0, there is no font information for the item in the corresponding `'DITL'`, and no data follows. If you specify a value of 1, there is font information for the item, and the rest of the structure is read. You can cause entries to be skipped by setting this value to 0.
- Dialog font flags. You can use one or more of these flag constants to specify which other fields in the dialog font table should be used; see "Dialog Font Flag Constants" (page 258).

- Font ID. If the `kDialogFontUseFontMask` bit is set to 1, then this element will contain an integer indicating the ID of the font family to use. See "Meta Font Constants" (page 103) for more information about the constants that you can specify. If this bit is set to 0, then the system default font is used.
- Font size. If a constant representing the system font, small system font, or small emphasized system font is specified in the Font ID field, this field is ignored. If the `kDialogFontUseSizeMask` bit is set, this field should contain an integer representing the point size of the text. If the `kDialogFontAddSizeMask` bit is set, this value will contain the size to add to the current point size of the text.
- Style. If the `kDialogFontUseFaceMask` bit is set, then this element should contain an integer specifying the text style to describe which styles to apply to the text. You can use one or more of the following `style` data type mask constants to specify font style:

Bit value	Style
0x00	Normal
0x01	Bold
0x02	Italic
0x04	Underline
0x08	Outline
0x10	Shadow
0x20	Condense
0x40	Extend

- Text mode. If the `kDialogFontUseModeMask` bit is set, then this element should contain an integer specifying how characters are drawn. See *Inside Macintosh: Imaging With QuickDraw* for a discussion of source transfer modes.
- Justification. If the `kDialogFontUseJustMask` bit is set, then this element should contain an integer specifying text justification (left, right, centered, or system-justified).
- Text color. If the `kDialogUseFontForeColorMask` bit is set, then this element should contain a color to use when drawing the text.
- Background color. If the `kDialogFontUseBackColorMask` bit is set, then this element should contain a color to use when drawing the background behind the text. In certain text modes, background color is ignored.
- Font name. If the `kDialogFontUseFontNameMask` bit is set, then this element should contain a Pascal string representing the font name to be used. This overrides the font ID.

Dialog Font Flag Constants

NEW WITH THE APPEARANCE MANAGER

You can set the following bits in the dialog font table resource (page 254) to specify fields in the dialog font table that should be used.

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

Constant descriptions

`kDialogFontNoFontStyle`
: If the `kDialogFontNoFontStyle` constant is used, no font style information is applied.

`kDialogFontUseFontMask`
: If the `kDialogFontUseFontMask` flag (bit 0) is set, the font ID specified in the Font ID field of the dialog font table is applied.

`kDialogFontUseFaceMask`
: If the `kDialogFontUseFaceMask` flag (bit 1) is set, the font style specified in the Style field of the dialog font table is applied.

`kDialogFontUseSizeMask`
: If the `kDialogFontUseSizeMask` flag (bit 2) is set, the font size specified in the Font Size field of the dialog font table is applied.

`kDialogFontUseForeColorMask`
: If the `kDialogFontUseForeColorMask` flag (bit 3) is set, the

text color specified in the Text Color field of the dialog font table is applied. This flag only applies to static text controls.

`kDialogFontUseBackColorMask`

If the `kDialogFontUseBackColorMask` flag (bit 4) is set, the background color specified in the Background Color field of the dialog font table is applied. This flag only applies to static text controls.

`kDialogFontUseModeMask`

If the `kDialogFontUseModeMask` flag (bit 5) is set, the text mode specified in the Text Mode field of the dialog font table is applied.

`kDialogFontUseJustMask`

If the `kDialogFontUseJustMask` flag (bit 6) is set, the text justification specified in the Justification field of the dialog font table is applied.

`kDialogFontUseAllMask`

If the `kDialogFontUseAllMask` constant is used, all flags in this mask will be set except `kDialogFontAddFontSizeMask` and `kDialogFontUseFontNameMask`.

`kDialogFontAddFontSizeMask`

If the `kDialogFontAddFontSizeMask` flag (bit 8) is set, the Dialog Manager will add a specified font size to the existing font size indicated in the Font Size field of the dialog font table resource.

`kDialogFontUseFontNameMask`

If the `kDialogFontUseFontNameMask` flag (bit 9) is set, the Dialog Manager will use the string in the Font Name field for the font name instead of a font ID.

The Dialog Color Table Resource

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and the `kDialogFlagsUseThemeBackground` feature bit of the extended dialog resource (page 249) is set, the entire dialog color table resource (`'dctb'`) is ignored. If the Appearance Manager is available, but the above is not true, the `wContent` field of the `'dctb'` resource is used, but all other fields are still ignored

The Alert Color Table Resource

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and the `kAlertFlagsUseThemeBackground` feature bit of the extended alert resource is set, the entire alert color table resource (`'actb'`) is ignored. If the Appearance Manager is available, but the `kAlertFlagsUseThemeBackground` bit is not set, the `wContent` field of the `'actb'` resource is used, but all other fields are still ignored.

The Item Color Table Resource

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and an embedding hierarchy is established in the dialog box, any item color table (`'ictb'`) resource information is ignored. The dialog font table (`'dftb'`) resource should be used instead of the item color table resource to specify the font settings for all dialog items in an embedding hierarchy.

If an embedding hierarchy is not established, the item color table resource can be used to set the font information for any editable text and static text dialog items, but the dialog font table resource will still be used for any controls in the dialog box.

Dialog Manager Functions

Creating Alerts

StandardAlert

NEW WITH THE APPEARANCE MANAGER

Displays a standard alert box.

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

`inAlertType` A constant indicating the type of alert box you wish to create; see "Alert Type Constants" (page 243).

`inError` A pointer to a Pascal string containing the primary error text you wish to display.

`inExplanation` A pointer to a Pascal string containing the secondary text you wish to display; secondary text is displayed in the small system font. Pass `nil` to indicate no secondary text.

`inAlertParam` A pointer to the standard alert structure; see "The Standard Alert Structure" (page 246). Pass `nil` to specify that you do not wish to your alert box to incorporate any of the features that the standard alert structure provides.

`outItemHit` A pointer to an integer that on output will contain a value indicating the alert button pressed; see "Alert Button Constants" (page 247).

function result A result code; see "Result Codes" (page 249).

DISCUSSION

The `StandardAlert` function displays an alert box based on the values you pass it. You can pass the error text you wish displayed in the `error` and `explanation` parameters, and customize the alert button text by filling in the appropriate fields of the standard alert structure passed in the `inAlertParam` parameter.

`StandardAlert` automatically resizes the height of a dialog box to fit all static text. It ignores alert stages and therefore provides no corresponding alert sounds.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

Alert

CHANGED WITH THE APPEARANCE MANAGER

Displays an alert box (or, if appropriate for the alert stage, plays an alert sound instead of or in addition to displaying the alert box).

```
pascal short Alert  (short alertID,
                     ModalFilterUPP modalFilter);
```

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

`modalFilter` A universal procedure pointer for a filter function that responds to events not handled by the `ModalDialog` (page 281) function. If you set this parameter to `nil`, the Dialog Manager uses the standard event filter function.

function result If no alert box is to be drawn at the current alert stage or the `'ALRT'` resource is not found, `Alert` returns –1; otherwise, it creates and displays the alert box and returns the item number of the control selected by the user; see "Alert Button Constants" (page 247).

DISCUSSION

The `Alert` function creates the alert defined in the specified alert resource and its corresponding extended alert resource. The function calls the current alert sound function 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` (page 281) function to get and handle most events for you.

The `Alert` 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.

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.

IMPORTANT

Your application should never draw its own default rings. Prior to Mac OS 8, the `Alert` function would only redraw the default button ring once and never redraw it on an update event. However, when Appearance is available, default rings do redraw when you call `Alert`.

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, call `AEInteractWithUser`; see *Inside Macintosh: Interapplication Communication*.

The Dialog Manager uses the system alert sound as the error sound unless you change it by calling the `ErrorSound` function .

SEE ALSO

`NoteAlert` (page 265).

`CautionAlert` (page 267).

`StopAlert` (page 264).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`Alert` only reads in the resource ID of an alert resource, not an extended alert resource.

StopAlert

CHANGED WITH THE APPEARANCE MANAGER

Displays an alert box with a stop icon in its upper-left corner (or, if appropriate for the alert stage, plays an alert sound instead of or in addition to displaying the alert box).

```
pascal short StopAlert (short alertID, ModalFilterUPP modalFilter);
```

`alertID` The resource ID of an alert resource and extended alert resource. The resource ID of both types of resources must be identical. If the alert resource is missing, the Dialog Manager returns to your application without creating the requested alert.

`modalFilter` A universal procedure pointer for a filter function that responds to events not handled by the `ModalDialog` (page 281) function. If you set this parameter to `nil`, the Dialog Manager uses the standard event filter function.

function result If no stop alert box is to be drawn at the current alert stage, `StopAlert` returns –1; otherwise, it creates and displays the alert box and returns the item number of the control selected by the user; see "Alert Button Constants" (page 247).

DISCUSSION

The `StopAlert` function is the same as the `Alert` function (page 262) except that, before drawing the items in the alert box, `StopAlert` draws the stop icon in the upper-left corner. The stop icon has resource ID 0, which you can also specify with the constant `stopIcon` . 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 resource ID 0.

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.

IMPORTANT

Your application should never draw its own default rings or alert icons. Prior to Mac OS 8, the `StopAlert` function would only redraw the alert icon and default button ring once and never redraw them on an update event. However, when Appearance is available, alert icons and default rings do redraw when you call `StopAlert`.

SEE ALSO

`NoteAlert` (page 265).

`CautionAlert` (page 267).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`StopAlert` only reads in the resource ID of an alert resource, not an extended alert resource.

NoteAlert

CHANGED WITH THE APPEARANCE MANAGER

Displays an alert box with a note icon in its upper-left corner (or, if appropriate for the alert stage, plays an alert sound instead of or in addition to displaying the alert box).

```
pascal short NoteAlert (
                    short alertID,
                    ModalFilterUPP modalFilter);
```

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

`modalFilter`

A universal procedure pointer for a filter function that responds to events not handled by the `ModalDialog` (page 281) function. If you set this parameter to `nil`, the Dialog Manager uses the standard event filter function.

function result If no alert box is to be drawn at the current alert stage, `NoteAlert` returns –1; otherwise, it creates and displays the alert box and returns the item number of the control selected by the user; see "Alert Button Constants" (page 247).

DISCUSSION

The `NoteAlert` function is the same as the `Alert` (page 262) function except that, before drawing the items in the alert box, `NoteAlert` draws the note icon in the upper-left corner. The note icon has resource ID 1, which you can also specify with the constant `noteIcon`. 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 resource ID 1.

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.

IMPORTANT

Your application should never draw its own default rings or alert icons. Prior to Mac OS 8, the `NoteAlert` function would only redraw the alert icon and default button ring once and never redraw them on an update event. However, when Appearance is available, alert icons and default rings do redraw when you call `NoteAlert`.

SEE ALSO

`CautionAlert` (page 267).

`StopAlert` (page 264).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`NoteAlert` only reads in the resource ID of an alert resource, not an extended alert resource.

CautionAlert

CHANGED WITH THE APPEARANCE MANAGER

Displays 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).

```
pascal short CautionAlert (
                    short alertID,
                    ModalFilterUPP modalFilter);
```

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

`modalFilter`
A universal procedure pointer for a filter function that responds to events not handled by the `ModalDialog` function (page 281). If you set this parameter to `nil`, the Dialog Manager uses the standard event filter function.

function result If no alert box is to be drawn at the current alert stage, `CautionAlert` returns –1; otherwise, it uses `NewDialog` to create and display the alert box and returns the item hit; see "Alert Button Constants" (page 247).

DISCUSSION

The `CautionAlert` function is the same as the `Alert` (page 262) function except that, before drawing the items in the alert box, `CautionAlert` draws the caution icon in the upper-left corner. The caution icon has resource ID 2, which you can also specify with the constant `kCautionIcon`. 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 resource ID 2.

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).

IMPORTANT

Your application should never draw its own default rings or alert icons. Prior to Mac OS 8, the `CautionAlert` function would only redraw the alert icon and default button ring once and never redraw them on an update event. However, when Appearance is available, alert icons and default rings do redraw when you call `CautionAlert`.

SEE ALSO

`NoteAlert` (page 265).

`StopAlert` (page 264).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`CautionAlert` only reads in the resource ID of an alert resource, not an extended alert resource.

Creating Dialog Boxes

GetNewDialog

CHANGED WITH THE APPEARANCE MANAGER

Creates a dialog box from a resource-based description.

```
pascal DialogPtr GetNewDialog (
                    short dialogID,
                    void *dStorage,
                    WindowPtr behind);
```

`dialogID`	The resource ID of a dialog resource and an extended dialog resource. The resource IDs for both resources must be identical. If the dialog resource is missing, the Dialog Manager returns to your application without creating the requested dialog box.
`dStorage`	A pointer to the memory for the dialog structure. If you set this parameter to `nil`, the Dialog Manager automatically allocates a nonrelocatable block in your application heap.
`behind`	A pointer to the window behind which the dialog box is to be placed on the desktop. Set this parameter to the window pointer `(WindowPtr)-1L` to bring the dialog box in front of all other windows.
function result	Returns a pointer to a dialog box. If none was created, returns `nil`.

DISCUSSION

The `GetNewDialog` function creates a dialog structure from information in a dialog resource and an extended dialog resource (if it exists) and returns a pointer to the dialog structure. You can use this pointer with Window Manager or QuickDraw functions 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 function `ShowWindow` to display the dialog box.

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

If you supply a dialog color table (`'dctb'`) resource with the same resource ID as the dialog resource, `GetNewDialog` uses `NewColorDialog` 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. However, if the Appearance Manager is available and the `kDialogFlagsUseThemeBackground` feature bit of the extended dialog resource is set, then the `'dctb'` resource is ignored and a color graphics port is created.

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 or you should release the resources yourself.

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.

As with all other windows, dialogs are created with an update region equal to their port rectangle. However, if the dialog's `'DLOG'` resource specifies that the dialog be made visible upon creation, the Dialog Manager draws the controls immediately and calls `ValidRgn` for each of their bounding rectangles. Other items are not drawn until the first update event for the dialog box is serviced.

If you need to display an alert box while your application is running in the background or is otherwise invisible to the user, call `AEInteractWithUser`; see *Inside Macintosh: Interapplication Communication*.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`GetNewDialog` does not read in the resource IDs of extended dialog resources or dialog font table resources.

NewFeaturesDialog

NEW WITH THE APPEARANCE MANAGER

Creates a dialog box from information passed in memory.

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

`inStorage`	A pointer to the memory for the dialog structure. If you set this parameter to `nil`, the Dialog Manager automatically allocates a nonrelocatable block in your application heap.
`inBoundsRect`	A pointer to a rectangle, given in global coordinates, that determines the size and position of the dialog box; these coordinates specify the upper-left and lower-right corners of the dialog box.
`inTitle`	A pointer to 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.
`inIsVisible`	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 function `ShowWindow`, described in "Displaying Windows" (page 227), to display it.
`inProcID`	The window definition ID for the type of dialog box, specified with constants defined by the Window Manager. Use the `kWindowModalDialogProc` constant to specify modal dialog boxes, the `kWindowDocumentProc` constant to specify modeless dialog boxes, and the `kWindowMovableModalDialogProc` constant to specify movable modal dialog boxes.
`inBehind`	A pointer to the window behind which the dialog box is to be placed on the desktop. Set this parameter to the window pointer `(WindowPtr)-1L` to bring the dialog box in front of all other windows.
`inGoAwayFlag`	A Boolean value. If `true`, specifies that an active modeless dialog box has a close box in its title bar.
`inRefCon`	A value that the Dialog Manager uses to set the `refCon` field of the dialog box's window structure. Your application may store any value here for any purpose. For example, your application can store a number that represents a dialog box type, or it can store a handle to a structure that maintains state information about the dialog box. You can use the Window Manager function `SetWRefCon` at any time to change this value in the dialog structure for a dialog box, and you can use the `GetWRefCon` function to determine its current value.

`inItemListHandle`
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.

`inFlags` An unsigned 32-bit mask specifying the dialog box's Appearance-compliant feature flags; see "Dialog Feature Flag Constants" (page 244). To establish an embedding hierarchy in a dialog box, pass `kDialogFlagsUseControlHierarchy` in the `inFlags` parameter.

function result Returns a pointer to the new dialog box. If it doesn't create a new dialog box, returns `nil`.

DISCUSSION

The `NewFeaturesDialog` function creates a dialog box without using `'DLOG'` or `'dlgx'` resources. Although the `inItemListHandle` parameter specifies an item list (`'DITL'`) resource for the dialog box, the corresponding dialog font table (`'dftb'`) resource is not automatically accessed. You must explicitly set the dialog box's control font style(s) individually.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

Manipulating Items in Dialog and Alert Boxes

GetDialogItemAsControl

NEW WITH THE APPEARANCE MANAGER

Returns the control handle for a dialog item in an embedding hierarchy.

```
pascal OSErr GetDialogItemAsControl (
                    DialogPtr inDialog,
                    SInt16 inItemNo,
                    ControlHandle *outControl);
```

`inDialog`	A pointer to a dialog structure.
`inItemNo`	A number corresponding to the position of an item in the dialog box's item list.
`outControl`	A pointer to a control handle that, on output, will refer to the embedded control.
function result	A result code; see "Result Codes" (page 249). The Control Manager result code `errItemNotControl` indicates that the dialog item is not a control.

DISCUSSION

When an embedding hierarchy is established, `GetDialogItemAsControl` produces a handle to the embedded controls (except Help items). It should be used instead of `GetDialogItem` (page 273) when an embedding hierarchy is established.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

GetDialogItem

CHANGED WITH THE APPEARANCE MANAGER

Gets a handle to a dialog item.

```
pascal void GetDialogItem (
                    DialogPtr theDialog,
                    short itemNo,
                    short *itemType,
                    Handle *item,
                    Rect *box);
```

`theDialog`	A pointer to a dialog structure.
`itemNo`	The item number (a number corresponding to the position of an item in the dialog box's item list resource); use `FindDialogItem` (page 276) to determine this value.

`itemType`	On output, a pointer to a dialog item constant identifying the item type of the item requested in the `itemNo` parameter.
`item`	A pointer to a handle that, on output, will refer to the item specified in the `itemNo` parameter or, for application-defined draw functions, a pointer (coerced to a handle) to the draw function.
`box`	On output, a pointer to the rectangle that specifies the display rectangle (described in coordinates local to the dialog box), for the item specified in the `itemNo` parameter.

DISCUSSION

The `GetDialogItem` function produces the item type, a handle to the item (or, for application-defined draw functions, the function pointer), and the display rectangle for a specified item in an item list resource. When a control hierarchy is present in the dialog, `GetDialogItem` can gets the appropriate information (for example, a text handle) from the controls. If you wish to get a control handle for a dialog item in an embedding hierarchy, see `GetDialogItemAsControl` (page 272).

You should call `GetDialogItem` before calling functions such as `SetDialogItemText` (page 280) that need a handle to a dialog item.

SEE ALSO

`SetDialogItem` (page 275).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

An embedding hierarchy cannot be established in a dialog box.

SetDialogItem

CHANGED WITH THE APPEARANCE MANAGER

Sets or changes information for a dialog item.

```
pascal void SetDialogItem (
                    DialogPtr theDialog,
                    short itemNo,
                    short itemType,
                    Handle item,
                    const Rect *box);
```

`theDialog` A pointer to a dialog structure.

`itemNo` A number corresponding to the position of an item in the dialog box's item list resource; see `FindDialogItem` (page 276).

`itemType` A dialog item constant identifying the item type of the item specified in the `itemNo` parameter. When an embedding hierarchy is established, only the `kItemDisableBit` constant is honored.

`item` Either a handle to the dialog item specified in the `itemNo` parameter or, for a custom dialog item, a pointer (coerced to a handle) to an application-defined item drawing function. When an embedding hierarchy is established, the `item` parameter is ignored unless you pass a universal procedure pointer to an application-defined item draw function.

`box` On output, the display rectangle (in local coordinates) for the item specified in the `itemNo` parameter. If you set the control rectangle on an item when an embedding hierarchy is present, `SetDialogItem` will move and resize the item appropriately for you.

DISCUSSION

The `SetDialogItem` function sets the item specified by the `itemNo` parameter for the specified dialog box. If an embedding hierarchy exists, however, you cannot change the type or handle of an item, although application-defined item drawing functions can still be set.

SEE ALSO

`GetDialogItem` (page 273).

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

An embedding hierarchy cannot be established in a dialog box, so `SetDialogItem` honors all values passed in the `item` and `itemType` parameters.

GetDialogKeyboardFocusItem

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and an embedding hierarchy is established, you should call the Control Manager function `GetKeyboardFocus` (page 148) instead of `GetDialogKeyboardFocusItem` to return the item number of the item in a dialog box that has keyboard focus.

FindDialogItem

CHANGED WITH THE APPEARANCE MANAGER

Determines the item number of an item at a particular location in a dialog box.

```
pascal short FindDialogItem (
                    DialogPtr theDialog,
                    Point thePt);
```

`theDialog` A pointer to a dialog structure.

`thePt` The point (in local coordinates) where the mouse-down event occurred.

function result When an embedding hierarchy is established, the `FindDialogItem` function returns the deepest control selected by the user corresponding to the point specified in the `thePt` parameter. When an embedding hierarchy does not exist, `FindDialogItem` performs a linear search of the item list resource and returns a number corresponding to the hit item's position

in the item list resource. For example, it returns 0 for the first item in the item list, 1 for the second, and 2 for the third. If the mouse is not over a dialog item, `FindDialogItem` returns –1.

DISCUSSION

The function `FindDialogItem` is useful for changing the cursor when the user moves the cursor over a particular item.

To get the proper item number before calling the `GetDialogItem` (page 273) function or the `SetDialogItem` (page 275) function, add 1 to the result of `FindDialogItem`, as shown here:

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

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

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

The `FindDialogItem` function always does a linear search, because embedding is not available when the Appearance Manager is not available.

MoveDialogItem

NEW WITH THE APPEARANCE MANAGER

Moves a dialog item to a specified location in a window.

```
pascal OSErr MoveDialogItem (
                    DialogPtr inDialog,
                    SInt16 inItemNo,
                    SInt16 inHoriz,
                    SInt16 inVert);
```

`inDialog` A pointer to a dialog structure.

`inItemNo` A signed 16-bit integer representing the number of the dialog item within the item list.

`inHoriz`	A signed 16-bit integer representing the horizontal coordinate to which the dialog item should be moved.
`inVert`	A signed 16-bit integer representing the vertical coordinate to which the dialog item should be moved.
function result	A result code; see "Result Codes" (page 249).

DISCUSSION

The `MoveDialogItem` function moves a dialog item to a specified location in a window. `MoveDialogItem` ensures that if the item is a control, the control rectangle and the dialog item rectangle (maintained by the Dialog Manager) are always the same.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

SizeDialogItem

NEW WITH THE APPEARANCE MANAGER

Sizes a dialog item.

```
pascal OSErr SizeDialogItem (
                    DialogPtr inDialog,
                    SInt16 inItemNo,
                    SInt16 inHeight,
                    SInt16 inWidth);
```

`inDialog`	A pointer to a dialog structure.
`inItemNo`	A signed 16-bit integer representing the dialog item number within the item list.
`inHeight`	A signed 16-bit integer representing the desired height (in pixels) of the dialog item's control rectangle.
`inWidth`	A signed 16-bit integer representing the desired width (in pixels) of the dialog item's control rectangle.

function result A result code; see "Result Codes" (page 249).

DISCUSSION

The `SizeDialogItem` function resizes a dialog item to a specified size. If the dialog item is a control, the control rectangle and the dialog item rectangle (maintained by the Dialog Manager) are always the same.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

AutoSizeDialog

NEW WITH THE APPEARANCE MANAGER

Automatically resizes static text fields and their dialog boxes to accommodate changed static text.

```
pascal OSErr AutoSizeDialog (DialogPtr inDialog);
```

`inDialog` A pointer to a dialog structure.

function result A result code; see "Result Codes" (page 249).

DISCUSSION

The `AutoSizeDialog` function is useful in situations such as localization, where the size of a static text field (and the dialog box that contains it) may need to be altered to accommodate an alteration in the size of the static text.

For each static text item `AutoSizeDialog` finds in the item list resource, it adjusts the static text field and the bottom of the dialog box window to accommodate the text. Any items below a static text field are moved down. If the dialog box is visible when this function is called, it is hidden, resized, and then shown. If the dialog box has enough room to show the text as is, no resizing is done.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

Handling Text in Alert and Dialog Boxes

SetDialogItemText

CHANGED WITH THE APPEARANCE MANAGER

Sets the text string for static text and editable text fields.

```
pascal void SetDialogItemText (
                    Handle item,
                    ConstStr255Param text);
```

`item` A handle to an editable text or static text field. When embedding is on, you should pass in the control handle produced by a call to the function `GetDialogItemAsControl` (page 272). If embedding is not on, pass in the handle produced by the `GetDialogItem` (page 273) function.

`text` A pointer to a string containing the text to display in the field.

DISCUSSION

The `SetDialogItemText` function sets and redraws text strings for static text and editable text fields. `SetDialogItemText` is useful for supplying a default text string—such as a document name—for an editable text field while your application is running.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

An embedding hierarchy cannot be established in a dialog box.

Handling Events in Dialog Boxes

ModalDialog

CHANGED WITH THE APPEARANCE MANAGER

Handles events while your application displays a modal or movable modal dialog box.

```
pascal void ModalDialog (
                    ModalFilterUPP modalFilter,
                    short *itemHit);
```

`modalFilter` A universal procedure pointer for an event filter function. For modal dialog boxes, you can specify `nil` if you want to use the standard event-handling function. For movable modal dialog boxes, you should specify your own event filter function.

`itemHit` A pointer to a short integer. After receiving an event involving an enabled item, `ModalDialog` produces a number representing the position of the selected item in the active dialog box's item list resource.

DISCUSSION

Call the `ModalDialog` function immediately after displaying a modal or movable modal dialog box. Your application should continue calling `ModalDialog` until the user dismisses your dialog.

For modal dialogs, the `ModalDialog` function repeatedly handles events 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` plays the system alert sound and gets the next event.

For movable modal dialogs, if the `kDialogFlagsHandleMovableModal` feature bit in the extended dialog resource is set, the `ModalDialog` function will handle all standard movable modal user interactions, such as dragging a dialog box by its title bar and allowing the user to switch into another application. However, a difference between the `ModalDialog` function's behavior with movable modal and modal dialogs is that, with movable modal dialogs, your event filter

function receives all events. If you want the Dialog Manager to assist you in handling events in movable modal dialog boxes, call `GetStdFilterProc` and `StdFilterProc`.

For events inside the dialog box, `ModalDialog` passes the event to the event filter function pointed to in the `modalFilter` parameter before handling the event. When the event filter returns `false`, `ModalDialog` handles the event. If the event filter function handles the event, returning `true`, `ModalDialog` performs no more event handling.

If you set the `modalFilter` parameter to `nil`, the standard event filter function is executed. The standard event filter function checks whether

- the user has pressed the Enter or Return key and, if so, returns the item number of the default button
- the user has pressed the Escape key or Command-period and, if so, returns the item number of the Cancel button
- the cursor is over an editable text box, and optionally changes the cursor to an I-beam whenever this is the case

If you set the `modalFilter` parameter to point to your own event filter function, that function can use the standard filter function to accomplish the above tasks. (To do so, you can call `GetStdFilterProc`, and dispatch the event to the standard filter function yourself, or you can call `StdFilterProc` , which obtains a `ModalFilterUPP` for the standard filter function and then dispatches the function.) Additionally, your own event filter function should also

- handle update events, so that background processes can receive processor time, and return `false`
- return `false` for all events that your event filter function doesn't handle

You can also use your event filter function to test for and respond to keyboard equivalents and more complex events—for instance, the user dragging the cursor within an application-defined item. You can use your same event filter function in most or all of your alert and modal dialog boxes.

If the event filter function does not handle the event (returning `false`), `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` produces 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` function 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` produces 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` produces 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

The `ModalDialog` function 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.

When `ModalDialog` calls the Control Manager function `TrackControl`, it does not allow you to specify the action function necessary for anything more complex than a button, radio button, or checkbox. If you need a more complex control, 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.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

`ModalDialog` only handles events for modal dialogs.

Defining Your Own Dialog Item Function

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and an embedding hierarchy is established in a dialog box, you should provide the user pane drawing function `MyUserPaneDrawProc` (page 189) instead of the user item drawing function `MyUserItemProc` to draw an application-defined control (a dialog item becomes a control in a dialog box with an embedding hierarchy).

You can provide other user pane application-defined functions to hit test, track, perform idle processing, handle keyboard, activate, and deactivate event processing, handle keyboard focus, and set the background color or pattern in a user pane control. For examples of how to write these functions, see "Defining Your Own User Pane Functions" (page 189).

Menu Manager Reference

Contents

This chapter describes the Menu Manager types, constants, resources, and functions that are affected by Mac OS 8, the Appearance Manager, or contextual menus.

- "Menu Manager Types and Constants" (page 287) describe Menu Manager types and constants, including structures. Result codes are included at the end of this section.
- "Menu Manager Resources" (page 293) describes the menu (`'MENU'`) resource, the extended menu (`'xmnu'`) resource, and the menu color table (`'mctb'`) resource.
- "Menu Manager Functions" (page 304) describes both Menu Manager functions and application-defined callback functions.

Menu Manager Types and Constants

Contextual Menu Gestalt Selector Constants

NEW WITH CONTEXTUAL MENUS

Before calling any contextual menu functions, your application should pass the selector `gestaltContextualMenuAttr` to the `Gestalt` function to determine whether contextual menu functions are available.

```
enum{
    gestaltContextualMenuAttr           = 'cmnu'
};
```

Constant description

`gestaltContextualMenuAttr`
The `Gestalt` selector passed to the `Gestalt` function to determine whether contextual menu functions are available. Produces a value whose bits you should test to determine whether the contextual menu functions are available.

The following values are the bit numbers with which you can test for the presence of contextual menu functions:

```
enum{
    gestaltContextualMenuPresent        = 0,
    gestaltContextualMenuTrapAvailable  = 1
};
```

Constant descriptions

`gestaltContextualMenuPresent`
If this bit is set, the contextual menu functions are available to PowerPC applications. If this bit is not set, these functions are not available to PowerPC applications.

`gestaltContextualMenuTrapAvailable`
If this bit is set, the contextual menu functions are available to 68K applications. If this bit is not set, these functions are not available to 68K applications.

Menu Definition IDs

CHANGED WITH THE APPEARANCE MANAGER

A **menu definition ID** is supplied to the menu resource (page 293) or a menu-creation function such as `NewMenu` to specify which menu definition function to use in creating the menu. The menu definition ID contains the resource ID of the menu definition function.

When mapping is enabled, the pre-Appearance menu definition ID `textmenuProc` will be mapped to `kMenuStdMenuProc`, its Appearance-compliant equivalent. For a discussion on how to enable mapping, see "Introduction to the Appearance Manager" (page 19).

```
enum {
    textmenuProc        = 0,
    kMenuStdMenuProc    = 63,
    kMenuStdMenuBarProc = 63
};
```

Constant descriptions

`textmenuProc` The menu definition ID for menus that are not Appearance-compliant.

`kMenuStdMenuProc` The menu definition ID for Appearance-compliant menus.

`kMenuStdMenuBarProc` The menu bar definition ID for Appearance-compliant menu bars.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Only the menu definition ID constant `textmenuProc` (or the definition ID for a custom menu definition function) is supported.

Contextual Menu Help Type Constants

NEW WITH CONTEXTUAL MENUS

You can pass these constants in the `inHelpType` parameter of the function `ContextualMenuSelect` (page 311) to specify the kind of help the application supports.

```
enum{
    kCMHelpItemNoHelp           = 0,
    kCMHelpItemAppleGuide       = 1,
    kCMHelpItemOtherHelp        = 2
};
```

Constant descriptions

`kCMHelpItemNoHelp` The application does not support any help. The Menu Manager will put an appropriate help string into the menu and disable it.

`kCMHelpItemAppleGuide` The application supports Apple Guide help. The Menu Manager will put the name of the main Guide file into the menu and enable it.

`kCMHelpItemOtherHelp` The application supports some other form of help. In this case, the application must also pass a valid string into the

inHelpItemString parameter of ContextualMenuSelect. This string will be the text of the help item in the menu, and the help item will be enabled.

Contextual Menu Selection Type Constants

NEW WITH CONTEXTUAL MENUS

These constants are returned in the outUserSelectionType parameter of the function ContextualMenuSelect (page 311) to specify what the user selected from the contextual menu.

```
enum{
    kCMNothingSelected      = 0,
    kCMMenuItemSelected     = 1,
    kCMShowHelpSelected     = 3
};
```

Constant descriptions

kCMNothingSelected The user did not choose an item from the contextual menu and the application should do no further processing of the event.

kCMMenuItemSelected
The user chose one of the application's items from the menu. The application can examine the outMenuID and outMenuItem parameters of ContextualMenuSelect to see what the menu selection was, and it should then handle the selection appropriately.

kCMShowHelpSelected
The user chose the Help item from the menu. The application should open an Apple Guide database to a section appropriate for the selection. If the application supports some other form of help, it should be presented instead.

Modifier Key Mask Constants

NEW WITH THE APPEARANCE MANAGER

You can use one or more of these mask constants in the modifier keys field of the `'xmnu'` resource (page 298) to determine which modifier key(s) must be pressed along with a character key to create a keyboard equivalent for selecting a menu item. These constants are also passed in and obtained by `SetMenuItemModifiers` (page 322) and `GetMenuItemModifiers` (page 323), respectively.

```
enum {
    kMenuCommandModifiers       = 0,
    kMenuShiftModifier          = (1 << 0),
    kMenuOptionModifier         = (1 << 1),
    kMenuControlModifier        = (1 << 2),
    kMenuNoCommandModifier      = (1 << 3)
};
```

Constant descriptions

`kMenuCommandModifiers`
: If no bit is set, only the Command key is used in the keyboard equivalent.

`kMenuShiftModifier`
: If this bit (bit 0) is set, the Shift key is used in the keyboard equivalent.

`kMenuOptionModifier`
: If this bit (bit 1) is set, the Option key is used in the keyboard equivalent.

`kMenuControlModifier`
: If this bit (bit 2) is set, the Control key is used in the keyboard equivalent.

`kMenuNoCommandModifier`
: If this bit (bit 3) is set, the Command key is not used in the keyboard equivalent.

Menu Icon Handle Constants

NEW WITH THE APPEARANCE MANAGER

These constants specify the handle of the icon attached to the menu item. They are passed in `SetMenuItemIconHandle` (page 318) and obtained by `GetMenuItemIconHandle` (page 319), respectively.

```
enum {
    kMenuNoIcon         = 0,
    kMenuIconType       = 1,
    kMenuShrinkIconType = 2,
    kMenuSmallIconType  = 3,
    kMenuColorIconType  = 4,
    kMenuIconSuiteType  = 5
};
```

Constant descriptions

`kMenuNoIcon` No icon.

`kMenuIconType` An `'ICON'` handle.

`kMenuShrinkIconType` A 32-by-32 `'ICON'` handle shrunk (at display time) to 16-by-16.

`kMenuSmallIconType` A `'SICN'` handle.

`kMenuColorIconType` A `'cicn'` handle.

`kMenuIconSuiteType` An icon suite handle.

The Menu Color Information Table Structure

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard menus, only the menu title color (RGB1) and menu item text color (RGB2) fields of the menu color information table resource are used. If you specify 0 in the `mctID` and `mctItem` fields of the menu color entries of the menu bar in the table, the other colors will be used for the menus and menu bars.

If you are creating your own custom menu definition function, all entries in the table can be used.

Result Codes

The most common result codes returned by Menu Manager functions are listed below.

`noErr`	0	No error
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough memory
`resNotFound`	–192	Unable to read resource
`hmHelpManagerNotInited`	–855	Help manager not set up

Menu Manager Resources

The Menu Resource

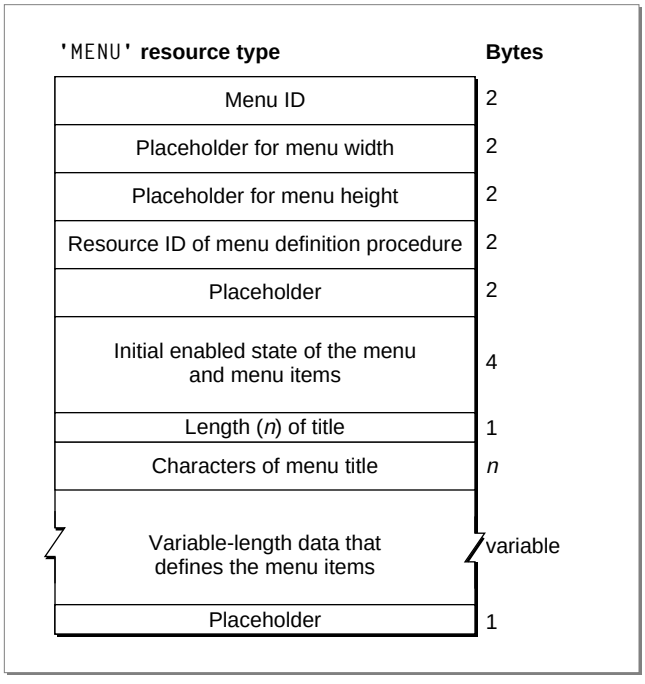
CHANGED WITH THE APPEARANCE MANAGER

You can provide descriptions of your menus in `'MENU'` resources and use the function `GetMenu` (page 307) or `GetNewMBar` to read the descriptions of your menus. After reading in the resource description, the Menu Manager stores the information about specific menus in menu structures. When you use a menu resource to define a menu, you should check for the presence of an extended menu resource with the same resource ID.

▲ **WARNING**
Menus in a resource must not be purgeable nor should they have the resource lock bit set. They must have resource ID numbers greater than 127. Do not define a "circular" hierarchical menu—that is, a hierarchical menu in which a submenu has a submenu whose submenu is a hierarchical menu higher in the chain.

Figure 5-1 shows the format of a compiled `'MENU'` resource.

Figure 5-1 Structure of a compiled menu (`'MENU'`) resource



A compiled version of a `'MENU'` resource contains the following elements:

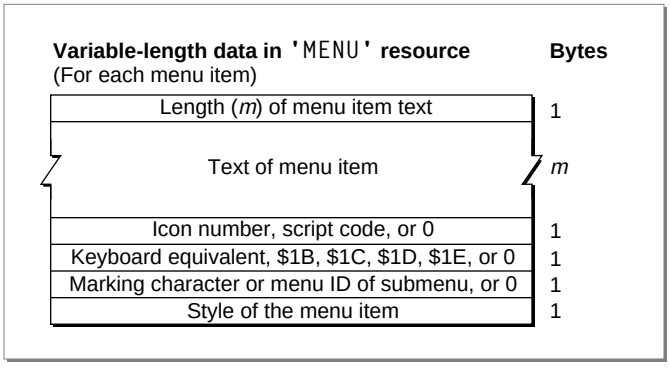
- Menu ID. Each menu in your application should have a unique menu ID (this can be the menu's resource ID). A negative value indicates that the menu (but not a submenu) belongs to a driver such as a desk accessory. A menu ID from 1 through 235 indicates a menu (or submenu) of an application; a menu ID from 236 through 255 indicates a submenu of a driver. Apple reserves the menu ID of 0.
- Placeholder (two integers containing 0) for the menu's width and height. After reading in the resource data, the Menu Manager requests the menu definition function to calculate the width and height of the menu and to store these values in the `menuWidth` and `menuHeight` fields of the menu structure.
- Resource ID of the menu's menu definition function; see "Menu Definition IDs" (page 288). If the integer 63 appears here, as specified by the

`kMenuStdMenuProc` constant in the Rez input file, the Menu Manager uses the standard Appearance-compliant menu definition function to manage the menu. If you provide your own menu definition function, its resource ID should appear in this field. After reading in the menu's resource data, the Menu Manager reads in the menu definition function, if necessary. The Menu Manager stores a handle to the menu definition function in the `menuProc` field of the menu structure.

- Placeholder (an integer containing 0).
- The initial enabled state of the menu and first 31 menu items. This is a 32-bit value, where bits 1–31 indicate if the corresponding menu item is disabled or enabled, and bit 0 indicates whether the menu as a whole is enabled or disabled. The Menu Manager automatically enables menu items greater than 31 when a menu is created.
- The length (in bytes) of the menu title.
- The title of the menu.
- Variable-length data that describes the menu items. If you provide your own menu definition function you can define and provide this data according to the needs of your function. The Menu Manager simply reads in the data for each menu item and stores it as variable data at the end of the menu structure. The menu definition function is responsible for interpreting the contents of the data. For example, the standard menu definition function interprets this data according to the description given in the following paragraphs.
- Placeholder (a byte containing 0) to indicate the end of the menu item definitions.

Figure 5-2 shows the variable-length data portion of a compiled `'MENU'` resource that uses the standard menu definition function.

Figure 5-2 The variable-length data that describes menu items as defined by the standard menu definition function



The variable-length data portion of a compiled version of a `'MENU'` resource that uses the standard menu definition function contains the following elements:

- Length (in bytes) of the menu item's text.
- Text of the menu item.
- A 1-byte field containing one of the following:
  - An icon number. The icon number is a number from 1 through 255 (or from 1 through 254 for small or reduced icons). The Menu Manager adds 256 to the icon number to generate the resource ID of the menu item's icon. If a menu item has an icon, you should also provide a `'cicn'`, `'SICN'`, or an `'ICON'` resource with the resource ID equal to the icon number plus 256. If you want the Menu Manager to reduce an `'ICON'` resource to the size of a small icon, you must also provide the value 0x1D in the keyboard equivalent field. If you provide a `'SICN'` resource, provide 0x1E in the keyboard equivalent field. Otherwise, the Menu Manager looks first for a `'cicn'` resource with the calculated resource ID and uses that icon.
  - A text encoding value. (Not recommended with Appearance.) If you want the Menu Manager to draw the item's text in a script other than the system script, specify the text encoding here and also provide 0x1C in the keyboard equivalent field. If the script system for the specified script is installed, the Menu Manager draws the item's text using that script.

- 0 (as specified by the `noicon` constant in a Rez input file) if the menu item doesn't contain an icon and uses the system script.

A menu item can have an icon or be drawn in a script other than the system script, but not both.

- Keyboard equivalent (specified as a 1-byte character). This can be enhanced with modifier key constants in the modifier keys field of the extended menu resource; see "The Extended Menu Resource" (page 298). In some cases, this field may take on one of the following values instead:
  - 0x1B (as specified by the constant `hierarchicalMenu` in a Rez input file) if the item has a submenu. (Not recommended with Appearance.)
  - 0x1C if the item uses a script other than the system script. (Not recommended with Appearance.)
  - 0x1D if you want the Menu Manager to reduce an `'ICON'` resource to the size of a small icon.
  - 0x1E if you want the Menu Manager to use an `'SICN'` resource for the item's icon.
  - 0 (as specified by the `nokey` constant in a Rez input file) if the item has neither a keyboard equivalent nor a submenu and uses the system script.

  The values 0x01 through 0x1A as well as 0x1F and 0x20 are reserved for use by Apple; your application should not use any of these reserved values in this field.
- A 1-byte field containing one of the following:
  - A marking character. Special marking characters are available to indicate the marks associated with a menu item.
  - The menu ID of the item's submenu. (Not recommended with Appearance.) Submenus of an application should have menu IDs from 1 through 235; submenus of a driver (such as a desk accessory) should have menu IDs from 236 through 255. If you choose a submenu, you must also set the keyboard equivalent field to 0x1B.
  - 0 (as specified by the `nomark` constant in a Rez input file) if the item has neither a mark nor a submenu.

  A menu item can have a mark or a submenu, but not both.
- Font style of the menu item. The constants `bold`, `italic`, `plain`, `outline`, and `shadow` can be used in a Rez input file to define their corresponding styles.

If you provide your own menu definition function, you should use the same format for your resource descriptions of menus as shown in Figure 5-1. You can

use the same format or one of your choosing to describe menu items. You can also use bits 1–31 of the `enableFlags` field of the menu structure as you choose; however, bit 0 must still indicate whether the menu is enabled or disabled.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

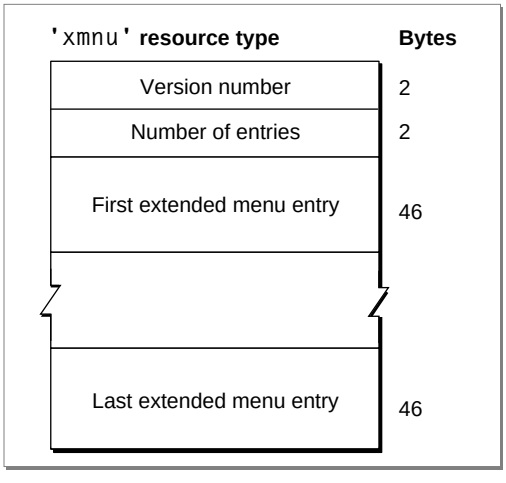
The menu resource, while identical in format, may take on different values in its variable-length data portion, depending on the type of information you want to display with your menu item.

The Extended Menu Resource

NEW WITH THE APPEARANCE MANAGER

After reading in a `'MENU'` resource, `GetMenu` (page 307) looks for an extended menu resource of type `'xmnu'` with the same resource ID. The **extended menu resource** allows you to create menus with modifier key keyboard glyphs and icons attached to menu items and Appearance-compliant menu backgrounds. The information is set for specified menu items (it is not necessary to create an extended menu entry for each item). At this point, the information can be purged or released. Figure 5-3 shows the format of a compiled `'xmnu'` resource.

Figure 5-3 Structure of a compiled extended menu (`'xmnu'`) resource

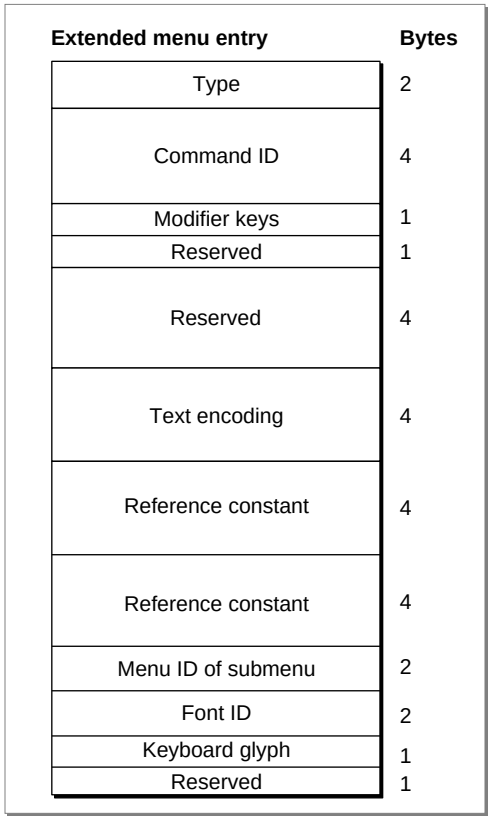


A compiled version of an `'xmnu'` resource contains the following elements:

- Version number. An integer specifying the version of the resource.
- Number of entries. An integer that specifies the number of entries in the resource. Each entry is an extended menu item structure.
- Extended menu item entries. A series of extended menu item structures, each of which consists of a type, command ID, modifier keys, text encoding, reference constants, menu ID of submenu, font ID, and keyboard glyph.

Figure 5-4 shows the format of an extended menu item entry.

Figure 5-4 Structure of an extended menu item entry



Each entry in a 'xmnu' resource corresponds to a menu item and contains the following:

- Type. An integer that specifies whether there is menu item information for the item in the 'MENU' entry. If this is 0, there is no information for the item in the corresponding 'MENU' entry, and the rest of the record is skipped. If this is 1, there is information for the item in the corresponding 'MENU' entry, and the rest of the record is read.
- Command ID. A unique value which you set to identify the menu item (instead of referring to it using the menu ID and item number). You can also call SetMenuItemCommandID (page 314) to set the command ID of a menu item. After a successful call to MenuSelect, MenuEvent (page 308), or MenuKey (page 310), you can call GetMenuItemCommandID (page 314) to determine its current value.
- Modifier keys. A mask that determines which modifier keys are used in a keyboard equivalent to select a menu item; see "Modifier Key Mask Constants" (page 291).
- Reserved. Set to 0.
- Reserved. Set to 0.
- Text encoding. A long integer which indicates the text encoding which your item text will use. Use currScript for the default text encoding. To change this value, call SetMenuItemTextEncoding (page 328). You can call GetMenuItemTextEncoding (page 329) to determine its current value. This should be used instead of setting a menu item's modifier key to 0x1C and its icon ID to the script code.
  - If you wish the text of the menu item to use the system script, this value should be -1. This should be used as the default.
  - If you wish the text of the menu item to use the current script, this value should be -2.
- Reference constant. Any value that an application wishes to store. To change this value, call SetMenuItemRefCon (page 324). You can call GetMenuItemRefCon (page 325) to determine its current value.
- Reference constant. Any additional value that an application wishes to store. To change this value, call SetMenuItemRefCon2 (page 326). You can call GetMenuItemRefCon2 (page 327) to determine its current value.
- Menu ID of submenu. A value between 1 and 235, identifying the application submenu.

- Font ID. An integer representing the ID of the font family. If this value is 0, then the system font ID is used.
- Keyboard glyph. A symbol representing a menu item's modifier key. In Appearance 1.0, if the value in this field is zero, the keyboard glyph uses the system font. In Appearance 1.0.1, if the value in this field is zero, the keyboard glyph uses the keyboard font; see Table 5-1 (page 302). Use of the keyboard font (rather than the system font) provides a consistent user interface across applications, since a modifier key's symbol will not change regardless of what system font is running. If the value in this field is nonzero, you can override the character code to be displayed with a substitute glyph.
- Reserved. Set to 0.

Table 5-1 Keyboard font character codes

Character code	Description
0x00	Null (always glyph 1)
0x01	Unassigned (reserved for 2 bytes)
0x02	Tab to the right key (for left-to-right script systems)
0x03	Tab to the left key (for right-to-left script systems)
0x04	Enter key
0x05	Shift key
0x06	Control key
0x07	Option key
0x08	Null (always glyph 1)
0x09	Space (always glyph 3) key
0x0A	Delete to the right key (for right-to-left script systems)
0x0B	Return key (for left-to-right script systems)
0x0C	Return key (for right-to-left script systems)
0x0D	Nonmarking return key
0x0E	Unassigned
0x0F	Pencil key
0x10	Downward dashed arrow key
0x11	Command key
0x12	Checkmark key
0x13	Diamond key
0x14	Apple logo key (filled)
0x15	Unassigned (paragraph in Korean)
0x16	Unassigned
0x17	Delete to the left key (for left-to-right script systems)
0x18	Leftward dashed arrow key
0x19	Upward dashed arrow key
0x1A	Rightward dashed arrow key

Character code	Description
0x1B	Escape key
0x1C	Clear key
0x1D	Unassigned (left double quotes in Japanese)
0x1E	Unassigned (right double quotes in Japanese)
0x1F	Unassigned (trademark in Japanese)
0x61	Blank key
0x62	Page up key
0x63	Caps lock key
0x64	Left arrow key
0x65	Right arrow key
0x66	Northwest arrow key
0x67	Help key
0x68	Up arrow key
0x69	Southeast arrow key
0x6A	Down arrow key
0x6B	Page down key
0x6C	Apple logo key (outline)
0x6D	Contextual menu key
0x6E	Power key
0x6F	F1 key
0x70	F2 key
0x71	F3 key
0x72	F4 key
0x73	F5 key
0x74	F6 key
0x75	F7 key
0x76	F8 key
0x77	F9 key
0x78	F10 key
0x79	F11 key

Character code	Description
0x7A	F12 key
0x87	F13 key
0x88	F14 key
0x89	F15 key
0x8A	Control key (ISO standard)

The Menu Color Information Table Resource

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available and you are using standard menus, only the menu title color (RGB1) and menu item text color (RGB2) fields of the menu color information table `('mctb')` resource are used. If you specify 0 in the `mctID` and `mctItem` fields in the menu color entries of the menu bar in the table, the other colors will be used for the menus and menu bars.

If you are creating your own custom menu definition function, all entries in the menu color information table can be used.

Menu Manager Functions

Initializing the Menu Manager

InitProcMenu

CHANGED WITH THE APPEARANCE MANAGER

Sets the `mbResID` field of the current menu list to the resource ID of a custom `'MBDF'` resource.

```
pascal void InitProcMenu (short resID);
```

`resID`	The resource ID of your application's menu bar definition function in the upper 13 bits of this parameter; the variant in the lower 3 bits. You must use a resource ID greater than 0x100. Resource IDs 0x000 through 0x100 are reserved for the use of Apple Computer, Inc.

DISCUSSION

If your application provides its own menu bar definition function, use the `InitProcMenu` function to associate your custom `'MBDF'` resource with the current menu list. In general, you should not use a custom menu bar definition unless absolutely necessary. `InitProcMenu` creates the current menu list if it hasn't already been created by a previous call to `InitMenus` .

You can also call `InitProcMenu` to bypass mapping of the pre-Appearance menu resource ID constant `textmenuProc` to its corresponding Appearance-compliant menu resource ID constant `kMenuStdMenuProc` when mapping is enabled. For information on mapping, see "Introduction to the Appearance Manager" (page 19).

SPECIAL CONSIDERATIONS

The resource ID of your application's menu bar definition function is maintained in the current menu list until your application next calls `InitMenus`; `InitMenus` initializes the `mbResID` field with the resource ID of the standard menu bar definition function. This can affect applications such as development environments that control other applications that may call `InitMenus`.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

Definition function mapping is not supported.

InitContextualMenus

NEW WITH CONTEXTUAL MENUS

Adds the application to the system registry of contextual menu clients.

```
pascal OSStatus InitContextualMenus (void);
```

function result A result code; see "Result Codes" (page 293).

DISCUSSION

Your application should call the `InitContextualMenus` function early in your startup code to register your application as a contextual menu client. If you do not register your application, some system-level functions may respond as though your application does not use contextual menus. Not registering your application may also cause `ProcessIsContextualMenuClient` (page 306) to return an incorrect value.

If you are a 68K application, you must pass the selector `gestaltContextualMenuAttr` to the `Gestalt` function before calling the `InitContextualMenus` function. If the `Gestalt` function returns a bit field with the `gestaltContextualTrapAvailable` bit set, `InitContextualMenus` can be called; see "Contextual Menu Gestalt Selector Constants" (page 287).

ProcessIsContextualMenuClient

NEW WITH CONTEXTUAL MENUS

Determines whether a given application is a contextual menu client.

```
pascal Boolean ProcessIsContextualMenuClient(ProcessSerialNumber* inPSN);
```

`inPSN` A pointer to the ID of the process containing the application.

function result A Boolean value; `true` if the application in the process uses contextual menus.

DISCUSSION

The `ProcessIsContextualMenuClient` function checks the system registry of contextual menu clients and returns `true` if the application in the given process supports contextual menus. However, the application must have been registered as a client using `InitContextualMenus` (page 305).

SEE ALSO

"Contextual Menu Gestalt Selector Constants" (page 287).

Creating Menus

GetMenu

CHANGED WITH THE APPEARANCE MANAGER

Creates a menu according to the specified menu and extended menu resources.

```
pascal MenuHandle GetMenu (short resourceID);
```

`resourceID` The resource ID of the menu and extended menu that defines the characteristics of the menu. (You usually use the same number for a menu's resource ID as the number that you specify for the menu ID in the menu resource.)

function result Returns a handle to the menu structure for the menu. You can use the returned menu handle to refer to this menu in most Menu Manager functions. If `GetMenu` is unable to read the menu or menu definition function from the resource file, `GetMenu` returns `nil`.

DISCUSSION

In addition to creating a menu, the `GetMenu` function also creates a menu structure for the menu. `GetMenu` reads the menu definition function into memory (if not already present) and stores a handle to the menu definition function in the menu structure. `GetMenu` does not insert the newly created menu into the current menu list.

Note
You typically use the `GetMenu` function only when you create submenus; you can create all your pull-down menus at once using the function `GetNewMBar`, and you can create pop-up menus using the standard pop-up menu button control definition function.

After reading the `'MENU'` resource (page 293), `GetMenu` searches for an extended menu resource and an `'mctb'` resource with the same resource ID as the `'MENU'` resource. If the specified `'mctb'` resource exists, `GetMenu` uses `SetMCEntries` to add the entries defined by the resource to the application's menu color

information table. If the `'mctb'` resource does not exist, `GetMenu` uses the default colors specified in the menu bar entry of the application's menu color information. If neither a menu bar entry nor a `'mctb'` resource exists, `GetMenu` uses the standard colors for the menu.

Storing the definitions of your menus in resources (especially menu titles and menu items) makes your application easier to localize.

▲ **WARNING**
Menus in a resource must not be purgeable nor should the resource lock bit be set. Do not define a "circular" hierarchical menu—that is, a hierarchical menu in which a submenu has a submenu whose submenu is a hierarchical menu higher in the chain.

SPECIAL CONSIDERATIONS

To release the memory associated with a menu that you created using `GetMenu`, first call `DeleteMenu` to remove the menu from the current menu list and to remove any entries for this menu in your application's menu color information table; then call `DisposeMenu` to dispose of the menu structure. After disposing of a menu, use `DrawMenuBar` to update the menu bar.

WHEN THE APPEARANCE MANAGER IS NOT AVAILABLE

The extended menu resource is not available.

Responding to the User's Choice of a Menu Command

MenuEvent

NEW WITH THE APPEARANCE MANAGER

Maps the keyboard equivalent character contained in the specified event structure to its corresponding menu and menu item.

```
pascal UInt32 MenuEvent (EventRecord* inEvent);
```

`inEvent`	A pointer to the event structure containing the keyboard equivalent.
function result	Returns a value that indicates the menu ID and menu item that the user chose. If the given character does not map to an enabled menu item in the current menu list, `MenuEvent` returns 0 in its high-order word and the low-order word is undefined.

DISCUSSION

The `MenuEvent` function does not distinguish between uppercase and lowercase letters. This allows a user to invoke a keyboard equivalent command, such as the Copy command, by pressing the Command key and "c" or "C". For consistency between applications, you should define the keyboard equivalents of your commands so that they appear in uppercase in your menus.

If the given character maps to an enabled menu item in the current menu list, `MenuEvent` highlights the menu title of the chosen menu, returns the menu ID in the high-order word of its function result, and returns the chosen menu item in the low-order word of its function result. After performing the chosen task, your application should unhighlight the menu title using the `HiliteMenu` function.

You should not define menu items with identical keyboard equivalents. The `MenuEvent` function scans the menus from right to left and the items from top to bottom. If you have defined more than one menu item with identical keyboard equivalents, `MenuEvent` returns the first one it finds.

The `MenuEvent` function first searches the regular portion of the current menu list for a menu item with a keyboard equivalent matching the given key. If it doesn't find one there, it searches the submenu portion of the current menu list. If the given key maps to a menu item in a submenu, `MenuEvent` highlights the menu title in the menu bar that the user would normally pull down to begin traversing to the submenu. Your application should perform the desired command and then unhighlight the menu title.

You shouldn't assign a Command–Shift–number key sequence to a menu item as its keyboard equivalent; Command–Shift–number key sequences are reserved for use as `'FKEY'` resources. Command–Shift–number key sequences are not returned to your application, but instead are processed by the Event Manager. The Event Manager invokes the `'FKEY'` resource with a resource ID that corresponds to the number that activates it.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

MenuKey

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, call `MenuEvent` (page 308) instead of `MenuKey` to map the keyboard equivalent character in the event structure to its corresponding menu and menu item.

IsShowContextualMenuClick

NEW WITH CONTEXTUAL MENUS

Determines whether a particular event could invoke a contextual menu.

```
pascal Boolean IsShowContextualMenuClick(const EventRecord* inEvent);
```

`inEvent` A pointer to the event structure that describes the event to examine.

function result Returns a Boolean value indicating whether or not a contextual menu should be displayed. If `true`, the contextual menu should be displayed; if `false`, not.

DISCUSSION

Before calling the `IsShowContextualMenuClick` function, you should call `InitContextualMenus` (page 305). If no error is returned, you can then call `IsShowContextualMenuClick`.

Applications should call `IsShowContextualMenuClick` when they receive non-null events. If `IsShowContextualMenuClick` returns `true`, your application should generate its own menu and Apple Event descriptor (`AEDesc`), and then call `ContextualMenuSelect` (page 311) to display and track the contextual menu, and then handle the user's choice.

If the mouse-down event did not invoke a contextual menu, then the application should check to see if the event occurred in the menu bar (using the

FindWindow function) and, if so, call MenuSelect to allow the user to choose a command from the menu bar.

SEE ALSO

"Contextual Menu Gestalt Selector Constants" (page 287).

ContextualMenuSelect

NEW WITH THE CONTEXTUAL MENUS

Displays a contextual menu.

```
pascal OSStatus ContextualMenuSelect (MenuHandle inMenuRef,
                    Point inGlobalLocation,
                    Boolean inReserved,
                    UInt32 inHelpType,
                    ConstStr255Param inHelpItemString,
                    const AEDesc* inSelection,
                    UInt32* outUserSelectionType,
                    SInt16* outMenuID,
                    UInt16* outMenuItem);
```

`inMenuRef` On input, a handle to a menu containing application commands to display. The caller creates this menu based on the current context, the mouse location, and the current selection (if it was the target of the mouse). If you pass `nil`, only system commands will be displayed. The menu should be added to the menu list as a pop-up menu (using the `InsertMenu` function).

`inGlobalLocation`
The location (in global coordinates) of the mouse near which the menu is to be displayed.

`inReserved` Reserved for future use. Pass `false` for this parameter.

`inHelpType` An identifier specifying the type of help to be provided by the application; see "Contextual Menu Help Type Constants" (page 289).

`inHelpItemString`
The string containing the text to be displayed for the help menu item. This string is unused unless you also pass the constant `kCMOtherHelp` in the `inHelpType` parameter.

`inSelection` On input, a pointer to an object specifier for the current selection. This allows the system to examine the selection and add special system commands accordingly. Passing a value of `nil` indicates that no selection should be examined, and most likely, no special system actions will be included.

`outUserSelectionType`
On output, a pointer to a value indicating what the user selected from the contextual menu; see "Contextual Menu Help Type Constants" (page 289).

`outMenuID` On output, a pointer to the menu ID of the chosen item, if `outUserSelectionType` is set to `kCMMenuItemSelected`.

`outMenuItem` On output, a pointer to the menu item chosen, if `outUserSelectionType` is set to `kCMMenuItemSelected`.

function result A result code; see "Result Codes" (page 293). The result code `userCanceledErr` indicates that the user did not select anything from the contextual menu and no further processing is needed; `outUserSelectionType` will be set to `kCMNothingSelected`.

DISCUSSION

If the `IsShowContextualMenuClick` function returns `true`, you should call the `ContextualMenuSelect` function after generating your own menu and preparing an Apple Event descriptor (`AEDesc`) that describes the item for which your application is displaying a contextual menu. This descriptor may contain an object specifier or raw data and will be passed to all contextual menu plug-ins.

The system will add other items before displaying the contextual menu, and it will remove those items before returning, leaving the menu in its original state.

After all the system commands are added, the contextual menu is displayed and tracked. If the user selects one of the system items, it is handled by the system and the call returns as though the user didn't select anything from the menu. If the user selects any other item (or no item at all), the Menu Manager passes back appropriate values in the parameters `outUserSelectionType`, `outMenuID`, and `outMenuItem`.

Your application should provide visual feedback indicating the item that was clicked upon. For example, a click on an icon should highlight the icon, while a click on editable text should not eliminate the current selection.

If the `outUserSelectionType` parameter contains `kCMMenuItemSelected`, you should look at the `outMenuID` and `outMenuItem` parameters to determine what menu item the user chose and handle it appropriately. If the user selected `kCMHelpItemSelected`, you should open the proper Apple Guide sequence or other form of custom help.

SEE ALSO

"Contextual Menu Gestalt Selector Constants" (page 287).

Manipulating and Accessing Menu Item Characteristics

SetItemCmd

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, you should call `SetMenuItemModifiers` (page 322), `SetMenuItemHierarchicalID` (page 317), and `SetMenuItemTextEncoding` (page 328) instead of `SetItemCmd` to set a menu item's keyboard equivalent and text encoding and to indicate that a menu item has a submenu.

SetItemMark

NOT RECOMMENDED WITH THE APPEARANCE MANAGER

When the Appearance Manager is available, you should call `SetMenuItemHierarchicalID` (page 317) instead of `SetItemMark` to set the menu ID of a menu item's submenu. However, you can still use `SetItemMark` to set the mark of a menu item.

SetMenuItemCommandID

NEW WITH THE APPEARANCE MANAGER

Sets a menu item's command ID.

```
pascal OSErr SetMenuItemCommandID (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    UInt32 inCommandID);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to set a command ID.
`inItem`	An integer representing the item number of the menu item for which you wish to set a command ID.
`inCommandID`	An integer representing the command ID that you wish to set.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

You can use a menu item's command ID as a position-independent method of signalling a specific action in an application.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemCommandID` (page 314).

GetMenuItemCommandID

NEW WITH THE APPEARANCE MANAGER

Gets a menu item's command ID.

```
pascal OSErr GetMenuItemCommandID (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    UInt32* outCommandID);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to get a command ID.
`inItem`	An integer representing the item number of the menu item for which you wish to get a command ID.
`outCommandID`	On output, a pointer to an integer representing the value of the item's command ID.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

After a successful call to `MenuSelect`, `MenuEvent` (page 308), or `MenuKey` (page 310), call the `GetMenuItemCommandID` function to get a menu item's command ID. You can use a menu item's command ID as a position-independent method of signalling a specific action in an application.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemCommandID` (page 314).

SetMenuItemFontID

NEW WITH THE APPEARANCE MANAGER

Sets the font for a menu item.

```
pascal OSErr SetMenuItemFontID (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16 inFontID);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to set the font.
`inItem`	An integer representing the item number of the menu item for which you wish to set the font.
`inFontID`	An integer representing the font ID that you wish to set.

function result A result code; see "Result Codes" (page 293).

DISCUSSION

The `SetMenuItemFontID` function enables you to set up a font menu with each item being drawn in the actual font.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemFontID` (page 316).

GetMenuItemFontID

NEW WITH THE APPEARANCE MANAGER

Gets a menu item's font ID.

```
pascal OSErr GetMenuItemFontID (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16* outFontID);
```

`inMenu` The handle to the menu structure of the menu item for which you wish to get a font ID.

`inItem` An integer representing the item number of the menu item for which you wish to get a font ID.

`outFontID` On output, a pointer to an integer representing the font ID for the menu item.

function result A result code; see "Result Codes" (page 293).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemFontID` (page 315).

SetMenuItemHierarchicalID

NEW WITH THE APPEARANCE MANAGER

Attaches a submenu to a menu item.

```
pascal OSErr SetMenuItemHierarchicalID (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16 inHierID)
```

`inMenu`	The handle to the menu structure of the menu item to which you wish to attach a submenu.
`inItem`	An integer representing the item number of the menu item to which you wish to attach a submenu.
`inHierID`	An integer representing the menu ID of the submenu you wish to attach. This menu should be inserted into the menu list by calling `InsertMenu`.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

The `SetMenuItemHierarchicalID` function should be called instead of setting the keyboard equivalent to 0x1B.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

GetMenuItemHierarchicalID

NEW WITH THE APPEARANCE MANAGER

Gets the menu ID of a specified submenu.

```
pascal OSErr GetMenuItemHierarchicalID (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16 *outHierID)
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to get the submenu's menu ID.
`inItem`	An integer representing the item number of the menu item for which you wish to get the submenu's menu ID.
`outHierID`	On output, a pointer to an integer representing the menu ID of the submenu.
function result	A result code; see "Result Codes" (page 293).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

SetMenuItemIconHandle

NEW WITH THE APPEARANCE MANAGER

Sets a menu item's icon.

```
pascal OSErr SetMenuItemIconHandle (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    MenuIconType inIconType,
                    Handle inIconHandle);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to set an icon.
`inItem`	An integer representing the item number of the menu item for which you wish to set an icon.
`inIconType`	The type of icon (`'ICON'`, `'cicn'`, `'SICN'`, or icon suite) you wish to attach, as specified by a menu icon handle constant (page 292).
`inIconHandle`	The handle to the icon you wish to attach to a menu item.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

The `SetMenuItemIconHandle` function sets the icon of a menu item with an icon handle instead of a resource ID. `SetMenuItemIconHandle` allows you to set icons of type `'ICON'`, `'cicn'`, `'SICN'`, as well as icon suites. To set resource-based icons for a menu item, call `SetItemIcon`.

▲ **WARNING**
Disposing of the menu will not dispose of the icon handles set by this function. To prevent memory leaks, your application should dispose of the icons when you dispose of the menu.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemIconHandle` (page 319).

GetMenuItemIconHandle

NEW WITH THE APPEARANCE MANAGER

Gets a handle to a menu item's icon.

```
pascal OSErr GetMenuItemIconHandle (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    MenuIconType outIconType,
                    Handle* outIconHandle);
```

`inMenu` The handle to the menu structure of the menu item with an icon for which you wish to get the handle.

`inItem` An integer representing the item number of the menu item with an icon for which you wish to get the handle.

`outIconType` On output, a menu item's icon type. If the menu item has no icon attached, this parameter will contain `kMenuNoIcon`.

`outIconHandle` On output, a pointer to a handle to the icon that is attached to your menu item; see "Menu Icon Handle Constants" (page 292). If the menu item has no icon suite attached, this parameter contains `nil`.

function result A result code; see "Result Codes" (page 293).

DISCUSSION

The `GetMenuItemIconHandle` function gets the icon handle and type of icon of the specified menu item. If you wish to get a resource-based menu item icon, call `GetItemIcon`.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemIconHandle` (page 318).

SetMenuItemKeyGlyph

NEW WITH THE APPEARANCE MANAGER

Substitutes a keyboard glyph for that normally displayed for a menu item's keyboard equivalent.

```
pascal OSErr SetMenuItemKeyGlyph (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16 inGlyph)
```

`inMenu` The handle to the menu structure of the menu item for which you wish to substitute a keyboard glyph.

`inItem` An integer representing the item number of the menu item for which you wish to substitute a keyboard glyph.

`inGlyph`	An integer representing the substitute glyph to display for characters that don't match their designated character codes. Pass 0 if you wish no substitution to occur. For a description of keyboard glyphs and a list of the keyboard font character codes, see "The Extended Menu Resource" (page 298).
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

The `SetMenuItemKeyGlyph` function overrides the character that would normally be displayed in a menu item's keyboard equivalent with a substitute keyboard glyph. This is useful if the keyboard glyph in the font doesn't match the actual character generated. For example, you might use this function to display function keys.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemKeyGlyph` (page 321).

GetMenuItemKeyGlyph

NEW WITH THE APPEARANCE MANAGER

Gets the keyboard glyph for a menu item's keyboard equivalent.

```
pascal OSErr GetMenuItemKeyGlyph (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16 *outGlyph)
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to get the keyboard glyph.
`inItem`	An integer representing the item number of the menu item for which you wish to get the keyboard glyph.

`outGlyph`	On output, a pointer to an integer representing the modifier key glyph. For a description of keyboard glyphs and a list of the keyboard font character codes, see "The Extended Menu Resource" (page 298).
function result	A result code; see "Result Codes" (page 293).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemIconHandle` (page 318).

SetMenuItemModifiers

NEW WITH THE APPEARANCE MANAGER

Sets the modifier key(s) that must be pressed with a character key to select a particular menu item.

```
pascal OSErr SetMenuItemModifiers (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16 inModifiers);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to set the modifier key(s).
`inItem`	An integer representing the item number of the menu item for which you wish to set the modifier key(s).
`inModifiers`	A value representing the modifier key(s) to be used in selecting the menu item; see "Modifier Key Mask Constants" (page 291).
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

You can call the `SetMenuItemModifiers` function to change the modifier key(s) you can include with a character key to create your keyboard equivalent. For example, you can change Command-*x* to Command-Option-Shift-*x*. By default, the Command key is always specified; however, you can remove the

Command key by setting the `kMenuNoCommand` flag in the modifier keys field of an extended menu item entry in the `'xmnu'` resource; see "The Extended Menu Resource" (page 298).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemModifiers` (page 323).

GetMenuItemModifiers

NEW WITH THE APPEARANCE MANAGER

Gets the modifier keys that must be pressed with a character key to select a particular menu item.

```
pascal OSErr GetMenuItemModifiers (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt16* outModifiers);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to get the modifier key(s).
`inItem`	An integer representing the item number of the menu item for which you wish to get the modifier key(s).
`outModifiers`	On output, a pointer to a mask representing the modifier keys that can be used in selecting the menu item; see "Modifier Key Mask Constants" (page 291).
function result	A result code; see "Result Codes" (page 293).

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemModifiers` (page 322).

SetMenuItemRefCon

NEW WITH THE APPEARANCE MANAGER

Sets application-specific information for a menu item.

```
pascal OSErr SetMenuItemRefCon (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt32 inRefCon);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to set information.
`inItem`	An integer representing the item number of the menu item for which you wish to set information.
`inRefCon`	An integer representing a reference constant with which you wish to set information.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

If you have any data you want to associate with a menu item, you can do so using the `SetMenuItemRefCon` function.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemRefCon` (page 325).

GetMenuItemRefCon

NEW WITH THE APPEARANCE MANAGER

Gets application-specific information for a menu item.

```
pascal OSErr GetMenuItemRefCon (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt32* outRefCon);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to get information.
`inItem`	An integer representing the item number of the menu item for which you wish to get information.
`outRefCon`	On output, a pointer to an integer representing a reference constant.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

If you have assigned any data to a given menu item using `SetMenuItemRefCon` function, you can read it using the `GetMenuItemRefCon` function.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemRefCon` (page 324).

SetMenuItemRefCon2

NEW WITH THE APPEARANCE MANAGER

Sets additional application-specific information for a menu item.

```
pascal OSErr SetMenuItemRefCon2 (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt32 inRefCon);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to set information.
`inItem`	An integer representing the item number of the menu item for which you wish to set information.
`inRefCon`	An integer representing a reference constant with which you wish to set information.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

If you have data you want to associate with a menu item in addition to that set with the `SetMenuItemRefCon` (page 324) function, you can do so using the `SetMenuItemRefCon2` function.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemRefCon2` (page 327).

GetMenuItemRefCon2

NEW WITH THE APPEARANCE MANAGER

Gets application-specific information for a menu item.

```
pascal OSErr GetMenuItemRefCon2 (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    SInt32* outRefCon);
```

`inMenu`	The handle to the menu structure of the menu item for which you wish to retrieve information.
`inItem`	An integer representing the item number of the menu item for which you wish to retrieve information.
`outRefCon`	On output, a pointer to an integer representing a reference constant.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

If you have assigned any data to a given menu item using `SetMenuItemRefCon2` function, you can read it using the `GetMenuItemRefCon` function.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemRefCon2` (page 326).

SetMenuItemTextEncoding

NEW WITH THE APPEARANCE MANAGER

Sets the text encoding for a menu item's text.

```
pascal OSErr SetMenuItemTextEncoding (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    TextEncoding inScriptID);
```

`inMenu`	A handle to the menu structure of the menu item whose text encoding you wish to set.
`inItem`	An integer representing the item number of the menu item whose text encoding you wish to set.
`inScriptID`	The script code that corresponds to the text encoding you wish to set.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

To set the text encoding for a menu item's text, call the `SetMenuItemTextEncoding` function instead of `SetItemCmd`. If a menu item has a command code of 0x1C when `SetMenuItemTextEncoding` is called, the values in the command and icon fields of the menu resource are cleared and replaced with the value in the `inScriptID` parameter of `SetMenuItemTextEncoding`.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`GetMenuItemTextEncoding` (page 329).

GetMenuItemTextEncoding

NEW WITH THE APPEARANCE MANAGER

Gets the text encoding used for a menu item's text.

```
pascal OSErr GetMenuItemTextEncoding (
                    MenuHandle inMenu,
                    SInt16 inItem,
                    TextEncoding* outScriptID);
```

`inMenu`	The handle to the menu structure of the menu item whose text encoding you wish to get.
`inItem`	An integer representing the item number of the menu item whose text encoding you wish to get.
`outScriptID`	On output, a pointer to the script code of the text encoding used in the menu item's text.
function result	A result code; see "Result Codes" (page 293).

DISCUSSION

If a menu item has a command code of 0x1C when `GetMenuItemTextEncoding` is called, `GetMenuItemTextEncoding` gets the value in the icon field of the menu item's menu resource.

SEE ALSO

"Appearance Manager Gestalt Selector Constants" (page 21).

`SetMenuItemTextEncoding` (page 328).

Defining Your Own Contextual Menu Plug-In

NEW WITH CONTEXTUAL MENUS

A contextual menu plug-in is a subclass of `AbstractCMPlugin : SOMObject`. It consists of four methods:

- `Initialize`, which is called when the Menu Manager creates a list of available plug-ins

- `ExamineContext`, which is called when the user activates a contextual menu
- `HandleSelection`, which handles a contextual menu selection chosen by the user
- `PostMenuCleanup`, which performs any necessary cleanup or deallocation after the contextual menu is dismissed.

Note
A contextual menu plug-in is implemented as a SOMObject object inside a shared library. (SOMObjects for the Mac OS platform is the Mac OS implementation of the System Object Model.) Typically your development environment can compile directly to a SOMObject object, so you do not need to create your own SOM interfaces.

Each subclass of the `AbstractCMPlugin` must have an extended `'cfrg'` resource, through which it identifies itself as a SOMObject object which derives from the `AbstractCMPlugin` class. See *Mac OS Runtime Architectures* for information about the extended `'cfrg'` resource.

In addition you must register the plug-in class as a SOMObject object so that the Menu Manager can instantiate it by name. Typically you can do this in a fragment's initialization function.

Listing 5-1 shows a sample initialization function that registers the plug-in.

Listing 5-1 Registering a contextual menu plug-in

```
pascal OSErr MyPluginInitialize(CFragInitBlockPtr init)
{
    /* If your compiler creates a default initialization function,*/
    /* you should call it here */

    /* Now register our class with SOM */
        somNewClass(MyPlugIn);

    return noErr;
}
```

The class declaration for a contextual menu definition plug-in is as follows:

```
class AbstractCMPlugin: SOMObject
{
    OSStatus Initialize(FSSpec *inFileSpec);
    OSStatus ExamineContext(AEDesc* inContextDescriptor,
                    SInt32 inTimeOutInTicks
                    AEDescList* ioCommandPairs,
                    Boolean* outNeedMoreTime);
    OSStatus HandleSelection(AEDesc* inContextDescriptor, SInt32
                    inCommandID);
    OSStatus PostMenuCleanup(void);
}
```

When writing your own contextual menu plug-in, you must follow this declaration and include the specified methods. The following sections describe these methods in detail.

Initialize

NEW WITH CONTEXTUAL MENUS

Performs any required plug-in initialization. If you write a contextual menu plug-in, you may include an `Initialize` method with the following form:

```
OSStatus Initialize (FSSpec *inFileSpec);
```

`inFileSpec` A pointer to a file system specification record for the file that contains the plug-in.

method result A result code. See "Result Codes" (page 293) for a list of possible values. If this value is not `noErr` then the Menu Manager does not use the plug-in.

DISCUSSION

The `Initialize` method is called when the Menu Manager builds its registry of available plug-ins (typically at system startup). You should use the `Initialize` method to check for available resources before the plug-in is actually required. To maintain a small memory footprint, the `Initialize` method should not allocate any memory, buffers, or so on. Instead, you should allocate memory as needed when examining the context or acting on the selection.

ExamineContext

NEW WITH CONTEXTUAL MENUS

Examines the context chosen by the user and determines possible menu commands appropriate to the context. If you write a contextual menu plug-in, it must contain an `ExamineContext` method with the following form:

```
OSStatus ExamineContext (AEDesc* inContextDescriptor,
                    SInt32 inTimeOutInTicks,
                    AEDescList* ioCommandPairs,
                    Boolean* outNeedMoreTime);
```

`inContextDescriptor`
The context chosen by the user. The Menu Manager passes this in the form of a pointer to an Apple Event descriptor. See *Inside Macintosh: Interapplication Communication* for information about the form of this descriptor. If there is no selection to examine, the pointer is `NULL`.

`inTimeOutInTicks`
The amount of time the plug-in is allowed to examine the context and create menu items.

`ioCommandPairs`
A pointer to an Apple Event descriptor list containing the commands allowed for this context.

`outNeedMoreTime`
Not currently used.

method result A result code. See "Result Codes" (page 293) for a list of possible values. If this value is not `noErr` then the Menu Manager does not use the plug-in in this case. However, it will call `ExamineContext` again the next time a contextual menu is invoked.

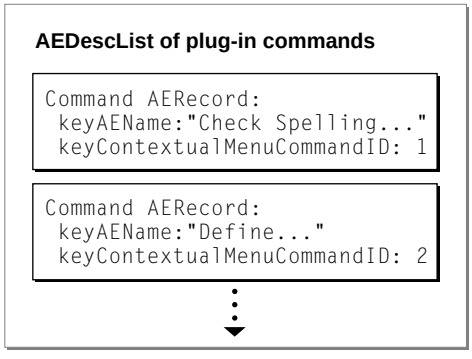
DISCUSSION

When a contextual menu is invoked, each module in the registry has its `ExamineContext` method called so it can inspect the context and add menu items as appropriate. After examining the context, the plug-in should then fill the `AEDescList` array with every command that it wants to add to the menu. This

AEDescList will be created and disposed of for the plug-in; it will be empty when the plug-in receives it.

Each menu command that the plug-in can perform on the selection is described in an AERecord with two keyword-specified descriptor records. The structure of the AERecord is shown in Figure 5-5.

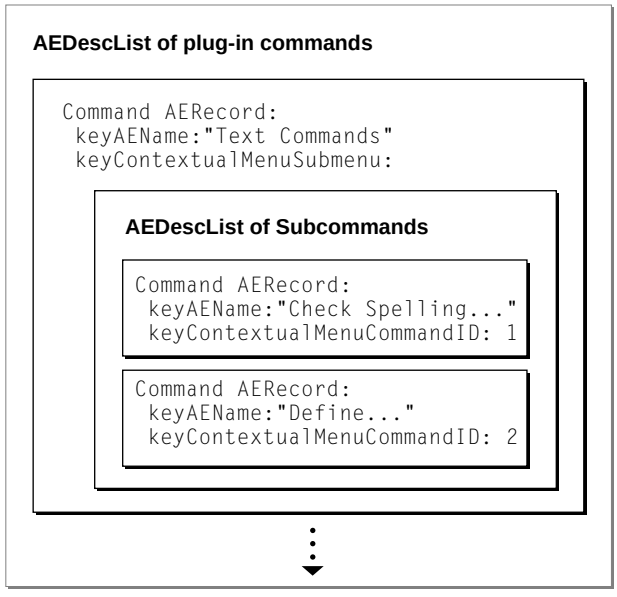
Figure 5-5 A menu command list in the AEDescList array



The first descriptor (keyAEName) is of typeIntlText and contains the text of the menu item to be added to the menu. The second descriptor (keyContextualMenuCommandID) is of type typeLongInteger and must contain a value specific to the plug-in that uniquely identifies this menu item.

If a plug-in wants to display a submenu for a particular menu item, it must use a variation of the AERecord used to describe a normal menu item. Figure 5-6 shows this variation.

Figure 5-6 A menu record showing submenus



The first descriptor (`keyAEName`) is the same, but the second descriptor uses a different keyword (`keyContextualMenuSubmenu`) and is of type `typeAEList`. It must contain an `AEDescList` with an `AERecord` for every command to be added to the submenu. Submenu items can themselves have submenus by recursively using this technique. The depth of the submenus is limited only by the constraints of the Menu Manager.

HandleSelection

NEW WITH CONTEXTUAL MENUS

Executes the contextual menu item chosen by the user. If you write a contextual menu plug-in, it must contain a `HandleSelection` method with the following form:

```
OSStatus HandleSelection(AEDesc* inContextDescriptor,
                         SInt32 inCommandID);
```

`inContextDescriptor`
: The context chosen by the user. This data is the same as that passed in the `inContextDescriptor` parameter for the `ExamineContext` method. The Menu Manager passes this in the form of a pointer to an Apple Event descriptor. See *Inside Macintosh: Interapplication Communication* for information about the form of this descriptor.

`inCommandID`
: A long integer assigned to the chosen menu item via the `keyContextualMenuCommandID` descriptor, passed in by the Menu Manager.

method result
: A result code. See "Result Codes" (page 293) for a list of possible values. If this value is not `noErr` then the Menu Manager does not use the plug-in in this case. However, it will call `HandleSelection` again the next time an action is selected.

DISCUSSION

If one of the plug-in's menu items is chosen, the Menu Manager calls the plug-in's `HandleSelection` method to execute the action. The plug-in should then perform the appropriate action.

PostMenuCleanup

NEW WITH CONTEXTUAL MENUS

Performs any necessary cleanup when the contextual menu is dismissed. If you write a contextual menu plug-in, it must contain a `PostMenuCleanup` method with the following form:

```
OSStatus PostMenuCleanup(void);
```

method result
: A result code. See "Result Codes" (page 293) for a list of possible values.

DISCUSSION

When a contextual menu is dismissed (regardless of whether or not the user made a selection), the Menu Manager calls each plug-in's `PostMenuCleanup`

method. The `PostMenuCleanup` method should do any necessary cleanup or memory deallocation. For example, a plug-in that allocated a buffer in the `ExamineContext` method should dispose of that buffer when `PostMenuCleanup` is called.

CHAPTER 6

Event Manager Reference

Contents

The Event Manager and Operating System Event Manager types and constants, resources, and functions in this chapter were not affected by Mac OS 8 or the Appearance Manager.

CHAPTER 7

Finder Interface Reference

Contents

This chapter describes the types, constants, and functions specific to the Finder Interface that are affected by Mac OS 8.

- "Finder Interface Types and Constants" (page 343) discusses the Finder Interface types and constants, including structures. Result codes are listed at the end of this section.
- "Finder Interface Functions" (page 355) describes functions that you can call to find and manipulate system-related folders and to get information on how the Finder routes files to the System Folder.

The changes to the Finder Interface that follow are related to the introduction of new folder management features with Mac OS 8. Folders now can be added to the System folder, or nested within other folders, and located via the `FindFolder` function. Previously, `FindFolder` only found folders that were immediately inside of the System Folder, and a few other special folders (such as the Trash folder and the System Folder itself). Now, once a folder is described in a **folder descriptor**, it can be contained within any other described folder and still be found by `FindFolder`. New features that allow you to get information on how the Finder routes files to the System Folder are also discussed.

▲ **WARNING**
All Folder Manager functions may move or purge memory and cannot be called at interrupt time.

Finder Interface Types and Constants

Folder Manager Gestalt Selector

CHANGED WITH MAC OS 8

Before calling any Folder Manager functions, your application should pass the selector `gestaltFindFolderAttr` to the `Gestalt` function to determine which Folder Manager functions are available.

```
enum {
    gestaltFindFolderAttr       ='fold'
};
```

Constant description

`gestaltFindFolderAttr`
: The `Gestalt` selector passed to determine whether the Folder Manager is available. Produces a value whose bits you should test to determine which Folder Manager functions are available:

```
enum {
    gestaltFindFolderPresent    = 0,
    gestaltFolderDescSupport    = 1
};
```

Constant descriptions

`gestaltFindFolderPresent`
: If this bit is set, the `FindFolder` (page 355) function is available. If this bit is clear, `FindFolder` is not available.

`gestaltFolderDescSupport`
: If this bit is set, the following Folder Manager functions are available: `AddFolderDescriptor`, `RemoveFolderDescriptor`, `GetFolderDescriptor`, `GetFolderTypes`, `IdentifyFolder`, `InvalidateFolderDescriptorCache`, `GetFolderName`, `GetFolderRoutings`, and `FindFolderRouting`. If this bit is clear, these functions are not available.

Folder Type Constants

CHANGED WITH MAC OS 8

You can pass these constants in the `folderType` parameter of the function `FindFolder` (page 355) to specify a folder on a particular volume.

```
enum {
    kSystemFolderType                   =   'macs',
    kDesktopFolderType                  =   'desk',
    kTrashFolderType                    =   'trsh',
```

```
    kWhereToEmptyTrashFolderType        =   'empt',
    kPrintMonitorDocsFolderType         =   'prnt',
    kStartupFolderType                  =   'strt',
    kShutdownFolderType                 =   'shdf',
    kFontsFolderType                    =   'font',
    kAppleMenuFolderType                =   'amnu',
    kControlPanelFolderType             =   'ctrl',
    kExtensionFolderType                =   'extn',
    kPreferencesFolderType              =   'pref',
    kTemporaryFolderType                =   'temp'
    kExtensionDisabledFolderType        =   'extD',
    kControlPanelDisabledFolderType     =   'ctrD',
    kSystemExtensionDisabledFolderType  =   'macD',
    kStartupItemsDisabledFolderType     =   'strD',
    kShutdownItemsDisabledFolderType    =   'shdD',
    kApplicationsFolderType             =   'apps',
    kDocumentsFolderType                =   'docs'
    kVolumeRootFolderType               =   'root',
    kChewableItemsFolderType            =   'flnt',
    kApplicationSupportFolderType       =   'asup',
    kTextEncodingsFolderType            =   'ftex',
    kStationeryFolderType               =   'odst',
    kOpenDocFolderType                  =   'odod',
    kOpenDocShellPlugInsFolderType      =   'odsp',
    kEditorsFolderType                  =   'oded',
    kOpenDocEditorsFolderType           =   'fodf',
    kOpenDocLibrariesFolderType         =   'odlb',
    kGenEditorsFolderType               =   'fedi',
    kHelpFolderType                     =   'fhlp',
    kInternetPlugInFolderType           =   'fnet',
    kModemScriptsFolderType             =   'fmod',
    kPrinterDescriptionFolderType       =   'ppdf',
    kPrinterDriverFolderType            =   'fprd',
    kScriptingAdditionsFolderType       =   'fscr',
    kSharedLibrariesFolderType          =   'flib',
    kVoicesFolderType                   =   'fvoc',
    kControlStripModulesFolderType      =   'sdev',
    kAssistantsFolderType               =   'astf',
    kUtilitiesFolderType                =   'utif',
    kAppleExtrasFolderType              =   'aexf',
    kContextualMenuItemsFolderType      =   'cmnu',
```

```
    kMacOSReadMesFolderType             =   'morf'
};
typedef OSType FolderType;
```

Constant descriptions

`kSystemFolderType`
Specifies a System Folder on a particular volume.

`kDesktopFolderType`
Specifies a Desktop Folder on a particular volume.

`kTrashFolderType` Specifies a single-user Trash folder on a particular volume.

`kWhereToEmptyTrashFolderType`
Specifies a shared Trash folder on a particular volume.

`kPrintMonitorDocsFolderType`
Specifies a Print Monitor Documents folder on a particular volume.

`kStartupFolderType`
Specifies a Startup Items folder on a particular volume.

`kShutdownFolderType`
Specifies a Shutdown Items folder on a particular volume.

`kFontsFolderType` Specifies a Fonts folder on a particular volume.

`kAppleMenuFolderType`
Specifies an Apple Menu Items folder on a particular volume.

`kControlPanelFolderType`
Specifies a Control Panels folder on a particular volume.

`kExtensionFolderType`
Specifies an Extensions folder on a particular volume.

`kPreferencesFolderType`
Specifies a Preferences folder on a particular volume.

`kTemporaryFolderType`
Specifies a Temporary folder on a particular volume.

`kExtensionDisabledFolderType`
Specifies a Disabled Extensions folder on a particular volume.

`kControlPanelDisabledFolderType`
Specifies a Disabled Control Panels folder on a particular volume.

`kSystemExtensionDisabledFolderType`
: Specifies a Disabled System Extension folder on a particular volume.

`kStartupItemsDisabledFolderType`
: Specifies a Disabled Startup Items folder on a particular volume.

`kShutdownItemsDisabledFolderType`
: Specifies a Disabled Shutdown Items folder on a particular volume.

`kApplicationsFolderType`
: Specifies an application folder on a particular volume.

`kDocumentsFolderType`
: Specifies a document folder on a particular volume.

`kVolumeRootFolderType`
: Specifies a root folder on a particular volume.

`kChewableItemsFolderType`
: Specifies an invisible folder on the system disk called "Cleanup at Startup" whose contents are deleted when the system is restarted, instead of merely being moved to the Trash. When the `FindFolder` function indicates this folder is available (by returning `noErr`), developers should usually use this folder for their temporary items, in preference to the Temporary Items folder.

`kApplicationSupportFolderType`
: This folder contains code and data files needed by third-party applications. These files should usually not be written to after they are installed. In general, files deleted from this folder remove functionality from an application, unlike files in the Preferences folder, which should be non-essential. One type of file that could be placed here would be plug-ins that the user might want to maintain separately from any application, such as for an image-processing application that has many "fourth-party" plug-ins that the user might want to upgrade separately from the host application. Another type of file that might belong in this folder would be application-specific data files that are not preferences, such as for a scanner application that needs to read description

files for specific scanner models according to which are currently available on the SCSI bus or network.

`kTextEncodingsFolderType`
Specifies the text encoding tables folder on a particular volume.

`kStationeryFolderType`
Specifies OpenDoc stationery folder on a particular volume.

`kOpenDocFolderType`
Specifies an OpenDoc root folder on a particular volume.

`kOpenDocShellPlugInsFolderType`
Specifies OpenDoc shell plug-ins in an OpenDoc folder on a particular volume.

`kEditorsFolderType`
Specifies an OpenDoc editors subfolder in a Mac OS folder on a particular volume.

`kOpenDocEditorsFolderType`
Specifies an OpenDoc subfolder in an editors folder on a particular volume.

`kOpenDocLibrariesFolderType`
Specifies an OpenDoc libraries folder on a particular volume.

`kGenEditorsFolderType`
Specifies a general editors folder on a particular volume.

`kHelpFolderType` Specifies a help folder on a particular volume.

`kInternetPlugInFolderType`
Specifies a folder of internet browser plug-ins on a particular volume.

`kModemScriptsFolderType`
Specifies a modem scripts folder (moved from the extensions folder) on a particular volume.

`kPrinterDescriptionFolderType`
Specifies a printer descriptions folder on a particular volume.

`kPrinterDriverFolderType`
Specifies a printer drivers folder on a particular volume.

`kScriptingAdditionsFolderType`
Specifies a text scripting additions folder on a particular volume.

`kSharedLibrariesFolderType`
Specifies a general shared libraries folder on a particular volume.

`kVoicesFolderType`
Specifies a MacinTalk folder on a particular volume.

`kControlStripModulesFolderType`
Specifies a Control Strip Modules folder on a particular volume.

`kAssistantsFolderType`
Specifies an Assistants folder (e.g., Mac OS Setup Assistant) on a particular volume.

`kUtilitiesFolderType`
Specifies a Utilities folder on a particular volume.

`kAppleExtrasFolderType`
Specifies an Apple Extras folder on a particular volume.

`kContextualMenuItemsFolderType`
Specifies a Contextual Menu Items folder on a particular volume.

`kMacOSReadMesFolderType`
Specifies a Mac OS Read Me's folder on a particular volume.

WHEN MAC OS 8 IS NOT AVAILABLE

Only the following folder types are available on a particular volume:

```
kSystemFolderType                 =   'macs',
kDesktopFolderType                =   'desk',
kTrashFolderType                  =   'trsh',
kWhereToEmptyTrashFolderType      =   'empt',
kPrintMonitorDocsFolderType       =   'prnt',
kStartupFolderType                =   'strt',
kShutdownFolderType               =   'shdf',
kFontsFolderType                  =   'font',
kAppleMenuFolderType              =   'amnu',
kControlPanelFolderType           =   'ctrl',
```

```
    kExtensionFolderType                =   'extn',
    kPreferencesFolderType              =   'pref',
    kTemporaryFolderType                =   'temp'
```

The Folder Descriptor Structure

NEW WITH MAC OS 8

This structure can be used to find existing folder descriptors and create new ones.

```
struct FolderDesc {
    size                    descSize;
    FolderType              foldType;
    FolderDescFlags         flags;
    FolderClass             foldClass;
    FolderType              foldLocation;
    OSType                  badgeSignature;
    OSType                  badgeType;
    UInt32                  reserved;
    Str63                   name
};
typedef struct FolderDesc FolderDesc;
typedef FolderDesc *FolderDescPtr;
```

Field descriptions

`descSize` Size (in bytes) of this structure.

`foldType` A constant of type `FolderType` that identifies the kind of target folder desired. See "Folder Type Constants" (page 344) for a list of possible folder types.

`flags` Flags indicating whether a folder is created during startup, if the folder name is locked, and if the folder created is invisible; see "Folder Descriptor Flag Constants" (page 351).

`foldClass` The class indicating whether the folder is relative to the parent folder or special; see "Folder Descriptor Class Constants" (page 352).

`foldLocation` For a relative folder, the `foldLocation` field specifies the `FolderType` of the parent folder of the target. For special

	folders, the location of the folder. See "Folder Descriptor Location Constants" (page 352).
`badgeSignature`	The `OSType` identifying the icon badge signature. Set to 0.
`badgeType`	The `OSType` identifying the icon badge type. Set to 0.
`reserved`	Reserved for use by system software.
`name`	A string specifying the name of the desired folder. For relative folders, this will be the exact name of the desired folder. For special folders, the actual target folder may have a different name than the name specified in the folder descriptor. For example, the System Folder is often given a different name, but it can still be located with `FindFolder` (page 355).

Folder Descriptor Flag Constants

NEW WITH MAC OS 8

These flags allow you to specify whether a folder is created during startup, if the name of the folder is locked when the folder is created, and if the folder created is invisible. Use these in the `flags` field of the folder descriptor structure (page 350). All other flag bits are reserved for future use by Apple Computer, Inc.

Set any combination of the following bits:

```
enum {
    kCreateFolderAtBoot         = 0x00000002,
    kFolderCreatedInvisible     = 0x00000004,
    kFolderCreatedNameLocked    = 0x00000008
};
typedef UInt32 FolderDescFlags;
```

Constant descriptions

`kCreateFolderAtBoot`
Folder created at boot if needed.

`kFolderCreatedInvisible`
Folder created as invisible.

`kFolderCreatedNameLocked`
Folder created with a locked name.

Folder Descriptor Class Constants

NEW WITH MAC OS 8

Constants of type `FolderClass` are used in the `foldClass` field to specify how the `foldLocation` field of the folder descriptor structure (page 350) should be interpreted.

IMPORTANT

Developers can only create new folder descriptors with a class of `kRelativeFolder`.

```
enum {
    kRelativeFolder = 'relf',
    kSpecialFolder  = 'spcf'
};
typedef OSType FolderClass;
```

Constant descriptions

`kRelativeFolder` This constant indicates that the `foldLocation` field contains the folder type of the parent folder, and the `name` field contains the name of the folder. Most folder descriptors are for relative folders.

`kSpecialFolder` Special folders are in locations hardwired into the Folder Manager, and can be found using special rules. Examples of special folders include a disk's root directory and System Folder. This constant indicates that the folder is located algorithmically, according to the constant supplied as the `foldLocation` field (`kBlessedFolder` or `kRootFolder`). Developers cannot create new folder descriptors of the `kSpecialFolder` class.

Folder Descriptor Location Constants

NEW WITH MAC OS 8

Constants of type `FolderClass` specify how the `foldLocation` field of the folder descriptor structure should be interpreted. There are two special folder locations that may be specified in the `foldLocation` field of the folder descriptor structure (page 350).

```
enum {
    kBlessedFolder      = 'blsf',
    kRootFolder         = 'rotf'
};
typedef OSType FolderLocation;
```

Constant descriptions

`kBlessedFolder` Constant used to locate the folder algorithmically if the `FolderClass` field is `kSpecialFolder`.

`kRootFolder` Constant used to locate the folder algorithmically if the `FolderClass` field is `kSpecialFolder`.

The Folder Routing Structure

NEW WITH MAC OS 8

The folder routing structure specifies the folder that files are routed to, based on the folder they are routed from.

IMPORTANT

Finder does not currently honor changes to the global folder routing list. Descriptions of some folder routing features are provided at this time for informational purposes only.

```
struct FolderRouting {
    Size                descSize;
    OSType              fileType;
    FolderType          routeFromFolder;
    FolderType          routeToFolder;
    RoutingFlags        flags
};
typedef struct FolderRouting FolderRouting;
typedef FolderRouting *FolderRoutingPtr;
```

Field descriptions

`descSize` Size (in bytes) of this structure.

`fileType` A constant of type `OSType` that describes the file type of the item to be routed.

`routeFromFolder`	The folder type identifying the folder from which an item will be routed. If an item is dropped on the folder specified in the `routeFromFolder` field, it will be routed to the folder described in the `routeToFolder` field.
`routeToFolder`	The folder type identifying the folder to which an item will be routed.
`flags`	These flags are reserved for future use by Apple Computer.

Result Codes

The most common result codes that can be returned by Finder Interface functions are listed below.

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`fnfErr`	–43	Folder not found.
`dupFNErr`	–48	File found instead of folder
`dirNFErr`	–120	Parent directory not found
`badFolderDescErr`	–4270	Invalid folder
`duplicateFolderDescErr`	–4271	Duplicate folders for a particular routing
`invalidFolderTypeErr`	–4273	Invalid folder name
`duplicateRoutingErr`	–4274	Same routing for two folders
`routingNotFoundErr`	–4275	No routing set up for folder passed in
`badRoutingSizeErr`	–4276	Incorrect `descSize` field of the folder routing structure

Finder Interface Functions

Finding Directories

FindFolder

CHANGED WITH MAC OS 8

Gets the location information used to gain access to the system-related directories.

```
pascal OSErr FindFolder (
                    short vRefNum,
                    OSType folderType,
                    Boolean createFolder,
                    short *foundVRefNum,
                    long *foundDirID);
```

`vRefNum` The volume reference number (or the constant `kOnSystemDisk` for the startup disk) of the volume on which you want to locate a directory; see "Folder Type Constants" (page 344).

`folderType` A four-character folder type, or a constant that represents the type, for the folder you want to find; see "Folder Type Constants" (page 344). Use the `kTrashFolderType` constant to locate the current user's Trash directory for a given volume—even one located on a file server. On a file server, you can use the `kWhereToEmptyTrashFolderType` constant to locate the parent directory of all logged-on users' Trash subdirectories.

`createFolder` Pass the constant `kCreateFolder` to create a directory if it does not already exist; otherwise, pass the constant `kDontCreateFolder`. Directories inside the System Folder are created only if the System Folder directory exists. The `FindFolder` function will not create a System Folder directory even if you specify the `kCreateFolder` constant in the

createFolder parameter. Passing kCreateFolder will also not create a parent folder; if the parent of the target folder does not already exist, attempting to create the target will fail.

foundVRefNum — On output, a pointer to the volume reference number for the volume containing the directory you specify in the folderType parameter.

foundDirID — On output, a pointer to the directory ID number for the directory you specify in the folderType parameter.

function result — A result code; see "Result Codes" (page 354). The result code fnfErr indicates that the type has not been found in the 'fld#' resource, or the disk doesn't have System Folder support, or the disk does not have desktop database support for Desktop Folder—in all cases, the folder has not been found. The result code dupFNErr indicates that a file has been found instead of a folder.

DISCUSSION

The FindFolder function can now be used to locate folders registered using the AddFolderDescriptor function.

For the folder type on the particular volume (specified, respectively, in the folderType and vRefNum parameters), the FindFolder function returns the directory's volume reference number through the foundVRefNum parameter and its directory ID through the foundDirID parameter. Each folder that can be found with FindFolder is described in a folder descriptor structure; see "The Folder Descriptor Structure" (page 350).

Those folders you're most likely to want to access are Preferences, Temporary Items, and Trash. For example, you might wish to check for the existence of a user's configuration file in Preferences, create a temporary file in Temporary Items, or, if your application runs out of disk storage when trying to save a file, check how much disk storage is taken by items in the Trash directory and report this to the user.

The specified folder used for a given volume might be located on a different volume in future versions of system software; therefore, do not assume the volume that you specify in vRefNum and the volume returned through foundVRefNum will be the same.

SPECIAL CONSIDERATIONS

Prior to Mac OS 8, the Finder identified the subdirectories of the System Folder, and their folder types, in a resource of type `'fld#'` located in the System file. While some backwards compatibility support for `'fld#'` remains, it has been superseded by the `'nfd#'` resource. As with `'fld#'`, you should not modify or rely on the contents of the `'nfd#'` resource in the System file. Instead, use only the `FindFolder` function to find the appropriate folders, and use the functions `AddFolderDescriptor` (page 357) and `RemoveFolderDescriptor` (page 359) to modify folder descriptors.

WHEN MAC OS 8 IS NOT AVAILABLE

The `FindFolder` function cannot be used to locate folders registered using the `AddFolderDescriptor` function. Also, prior to Mac OS 8, the Finder identified the subdirectories of the System Folder, and their folder types, in a resource of type `'fld#'` located in the System file. Do not modify or rely on the contents of the `'fld#'` resource in the System file; use only the `FindFolder` function to find the appropriate directories.

Manipulating Folder Descriptors

AddFolderDescriptor

NEW WITH MAC OS 8

Copies the supplied information into a new folder descriptor entry in the system folder list.

```
pascal OSErr AddFolderDescriptor (
                    FolderType foldType,
                    FolderDescFlags flags,
                    FolderClass foldClass,
                    FolderLocation foldLocation,
                    OSType badgeSignature,
                    OSType badgeType,
                    ConstStr63Param name,
                    Boolean replaceFlag);
```

`foldType`	A constant identifying the type of the folder which you wish the Folder Manager to be able to find. See "Folder Type Constants" (page 344).
`flags`	Set these flags to indicate whether a folder is created during startup, if the folder name is locked, and if the folder is created invisible; see "Folder Descriptor Flag Constants" (page 351).
`foldClass`	The class, indicating whether the folder is relative to the parent folder or special, of the folder which you wish the Folder Manager to be able to find; see "Folder Descriptor Class Constants" (page 352).
`foldLocation`	For a relative folder, specify the folder type of the parent folder of the target. For a special folder, specify the location of the folder; see "Folder Descriptor Location Constants" (page 352).
`badgeSignature`	The `OSType` identifying the icon badge signature. Set to 0.
`badgeType`	The `OSType` identifying the icon badge type. Set to 0.
`name`	For a relative folder, specify the name of the target folder. For a special folder, the actual target folder may have a different name than the name specified in the folder descriptor. For example, the System Folder is often given a different name, but it can still be located with `FindFolder` (page 355).
`replaceFlag`	A Boolean value indicating whether you wish to replace a folder routing that already exists. This prevents the return code `duplicateRoutingErr` from being returned. If `true`, it replaces the folder to which the item is being routed. If `false`, it leaves the folder to which the item is being routed.
function result	A result code; see "Result Codes" (page 354). The result code indicates that a folder descriptor is already installed with the specified folder type and `replaceFlag` is false.

DISCUSSION

The `AddFolderDescriptor` function copies the supplied information into a new descriptor entry in the system folder list. You need to provide folder descriptors for each folder you wish the Folder Manager to be able to find. For example, a child folder located in a parent folder needs to have a descriptor created both for it and its parent folder, so that the child can be found.

SPECIAL CONSIDERATIONS

Before calling the `AddFolderDescriptor` function, you must pass the selector `gestaltFindFolderAttr` to the `Gestalt` function. If the `gestaltFolderDescSupport` bit is set, `AddFolderDescriptor` is available.

RemoveFolderDescriptor

NEW WITH MAC OS 8

Removes the specified folder descriptor entry from the system folder list.

```
pascal OSErr RemoveFolderDescriptor (FolderType foldType);
```

`foldType` A constant identifying the type of the folder for which you wish to remove a descriptor. See "Folder Type Constants" (page 344).

function result A result code; see "Result Codes" (page 354).

DISCUSSION

Once a folder descriptor has been removed, `FindFolder` will no longer be able to locate the folder type.

SPECIAL CONSIDERATIONS

Before calling the `RemoveFolderDescriptor` function, you must pass the selector `gestaltFindFolderAttr` to the `Gestalt` function. If the `gestaltFolderDescSupport` bit is set, `RemoveFolderDescriptor` is available.

GetFolderDescriptor

NEW WITH MAC OS 8

Gets the folder descriptor information for the specified folder type from the global descriptor list.

```
pascal OSErr GetFolderDescriptor (
                    FolderType foldType,
                    Size descSize,
                    FolderDesc *foldDesc);
```

`foldType` A constant identifying the type of the folder for which you wish to get descriptor information. See "Folder Type Constants" (page 344).

`descSize` Input the size (in bytes) of the folder descriptor structure.

`foldDesc` On input, a pointer to the folder descriptor structure. On output, a pointer to a filled-out folder descriptor structure.

function result A result code; see "Result Codes" (page 354).

DISCUSSION

The `GetFolderDescriptor` function provides a pointer to a filled-out folder descriptor structure.

SPECIAL CONSIDERATIONS

Before calling the `GetFolderDescriptor` function, you must pass the selector `gestaltFindFolderAttr` to the `Gestalt` function. If the `gestaltFolderDescSupport` bit is set, `GetFolderDescriptor` is available.

GetFolderTypes

NEW WITH MAC OS 8

Gets the folder types from the global descriptor list.

```
pascal OSErr GetFolderTypes (
                    UInt32 requestedTypeCount,
                    UInt32 *totalTypeCount,
                    FolderType *theTypes);
```

`requestedTypeCount`
On input, the number of folder types that can fit in the buffer pointed to by the `theTypes` parameter.

`totalTypeCount`
On output, a pointer to the total number of folder types in the list. The `totalTypeCount` parameter may produce a value that is larger or smaller than that of the `requestedTypeCount` parameter. If `totalTypeCount` is equal to or smaller than the value passed in for `requestedTypeCount` and the value produced by the `theTypes` parameter is non-`nil`, then all folder types were returned to the caller.

`theTypes` On output, a pointer to a buffer containing a list of the folder types for the installed descriptors. You can step through the list and call `GetFolderDescriptor` for each folder type. You should usually pass a non-`nil` pointer to a buffer; you might pass `nil` only if you wanted to know the number of descriptors installed in the system's global list, rather than the actual folder types of those descriptors.

function result A result code; see "Result Codes" (page 354).

SPECIAL CONSIDERATIONS

Before calling the `GetFolderType` function, you must pass the selector `gestaltFindFolderAttr` to the `Gestalt` function. If the `gestaltFolderDescSupport` bit is set, `GetFolderType` is available.

IdentifyFolder

NEW WITH MAC OS 8

Gets the folder type for the specified folder.

```
pascal OSErr IdentifyFolder (
                    short vRefNum,
                    long dirID,
                    FolderType *foldType);
```

`vRefNum`	The volume reference number (or the constant `kOnSystemDisk` for the startup disk) of the volume containing the folder for which you wish the type to be identified.
`dirID`	The directory ID number for the folder for which you wish the type to be identified.
`foldType`	On output, a pointer to the folder type of the folder with the specified `vRefNum` and `dirID` parameters.
function result	A result code; see "Result Codes" (page 354).

DISCUSSION

The folder type is identified for the folder specified by the `vRefNum` and `dirID` parameters, if such a folder exists. `IdentifyFolder` may take several seconds to complete. Note that there may be multiple folder descriptors that map to an individual folder; `IdentifyFolder` returns the folder type of one of those descriptors.

SPECIAL CONSIDERATIONS

Before calling the `IdentifyFolder` function, you must pass the selector `gestaltFindFolderAttr` to the `Gestalt` function. If the `gestaltFolderDescSupport` bit is set, `IdentifyFolder` is available.

GetFolderName

NEW WITH MAC OS 8

Gets the name of the specified folder.

```
pascal OSErr GetFolderName (
                    short vRefNum,
                    OSType foldType,
                    short *foundVRefNum,
                    Str63 name);
```

`vRefNum`	The volume reference number (or the constant `kOnSystemDisk` for the startup disk) of the volume containing the folder for which you wish the name to be identified.
`foldType`	A constant identifying the type of the folder for which you wish the name to be identified. See "Folder Type Constants" (page 344).
`foundVRefNum`	On output, a pointer to the volume reference number for the volume containing the folder you specify in the `foldType` parameter.
`name`	On output, a string containing the title of the folder specified in the `foldType` and `vRefNum` parameters.
function result	A result code; see "Result Codes" (page 354).

DISCUSSION

The `GetFolderName` function gets the name of the folder in the folder descriptor, not of the folder on the disk. The names may differ for a few special folders such as the System Folder. For relative folders, however, the actual name is always returned. You typically do not need to call this function.

SPECIAL CONSIDERATIONS

Before calling the `GetFolderName` function, you must pass the selector `gestaltFindFolderAttr` to the `Gestalt` function. If the `gestaltFolderDescSupport` bit is set, `GetFolderName` is available.

InvalidateFolderDescriptorCache

NEW WITH MAC OS 8

Invalidates the cache of results from the Folder Manager's previous calls to the `FindFolder` function in order to force the Folder Manager to reexamine the disk when `FindFolder` is called again on the specified directory ID or volume reference number.

```
pascal OSErr InvalidateFolderDescriptorCache (
                    short vRefNum,
                    long dirID);
```

`vRefNum` The volume reference number (or the constant `kOnSystemDisk` for the startup disk) of the volume containing the folder for which you wish the descriptor cache to be invalidated. Pass 0 in this parameter to completely invalidate all folder cache information.

`dirID` The directory ID number for the folder for which you wish the descriptor cache to be invalidated. Pass 0 to invalidate the cache for all folders on the specified disk.

function result A result code; see "Result Codes" (page 354).

DISCUSSION

The `InvalidateFolderDescriptorCache` function takes a volume reference number and a directory ID and searches to see if it is currently referred to as a cached target of folder resolution. If it is found, it is removed as the cached value, but the folder descriptor is otherwise left unchanged. You should not normally need to call `InvalidateFolderDescriptorCache`.

SPECIAL CONSIDERATIONS

Before calling the `InvalidateFolderDescriptorCache` function, you must pass the selector `gestaltFindFolderAttr` to the `Gestalt` function. If the `gestaltFolderDescSupport` bit is set, `InvalidateFolderDescriptorCache` is available.

Routing Files

This section describes functions that can be used to get information on how the Finder routes files to the System Folder.

IMPORTANT

Finder does not currently honor changes to the global folder routing list. Descriptions of some folder routing features are provided at this time for informational purposes only.

GetFolderRoutings

NEW WITH MAC OS 8

Gets folder routing information from the global routing list.

```
pascal OSErr GetFolderRoutings (
                    UInt32 requestedRoutingCount,
                    UInt32 *totalRoutingCount,
                    Size routingSize,
                    FolderRouting *theRoutings);
```

`requestedRoutingCount`
On input, the number of folder routing structures that can fit in the buffer pointed to by the `theRoutings` parameter.

`totalRoutingCount`
On output, a pointer to the number of folder routing structures in the global list. If this value is less than or equal to `requestedRoutingCount`, all folder routing structures were returned to the caller.

`routingSize` On input, the size (in bytes) of the folder routing structure; see "The Folder Routing Structure" (page 353).

`theRoutings` Pass a pointer to a buffer containing one or more folder routing structures on output. If `nil` was passed, then only the total count of types will be returned in the `totalRoutingCount` parameter.

function result A result code; see "Result Codes" (page 354).

SPECIAL CONSIDERATIONS

Finder does not currently honor changes to the global folder routing list. Descriptions of some folder routing features are provided at this time for informational purposes only.

FindFolderRouting

NEW WITH MAC OS 8

Finds the destination folder from a matching folder routing structure for the specified file.

```
pascal OSErr FindFolderRouting (
                    OSType fileType,
                    FolderType routeFromFolder,
                    FolderType *routeToFolder,
                    RoutingFlags *flags);
```

`fileType` The file type specified in the appropriate folder routing structure for the file for which you wish to find a destination folder.

`routeFromFolder`
The folder type of the "from" folder for which you wish to find a "to" folder. An item dropped on the folder specified in this parameter will be routed to the folder specified in the `routeToFolder` parameter.

`routeToFolder` On output, a pointer to the folder type of the destination folder.

`flags` Pass a pointer to a variable to contain the output produced by the `flags` parameter, or `nil` for no response. Because no routing flags are currently defined, 0 will be returned in this parameter. The `flags` field of the folder routing structure is reserved for future use by Apple Computer, Inc.

function result A result code; see "Result Codes" (page 354).

DISCUSSION

Both the file type and the folder type specified must match those of a folder routing structure in the global routing list for the `FindFolderRouting` function to succeed.

SPECIAL CONSIDERATIONS

The system initializes the Folder Manager's routing tables with a resource of type `'nrt#'` located in the System file. You should not modify or rely on the contents of the `'nrt#'` resource in the System file; use only the `FindFolderRouting` function to find the appropriate folder routing information.

Finder does not currently honor changes to the global folder routing list. Descriptions of some folder routing features are provided at this time for informational purposes only.

AddFolderRouting

NEW WITH MAC OS 8

Adds a folder routing structure to the global routing list.

SPECIAL CONSIDERATIONS

Finder does not currently honor changes to the global routing list.

RemoveFolderRouting

NEW WITH MAC OS 8

Removes a folder routing structure from the global routing list.

SPECIAL CONSIDERATIONS

Finder does not currently honor changes to the global routing list.

Version History

This document has had the following releases:

Table A-1 *Mac OS 8 Toolbox Reference* Revision History

Version	Notes
Dec. 10, 1997	The following corrections were made: "Drawing Appearance-Compliant Controls" (page 35). Noted that an error would be returned by those control-drawing functions that may not be passed the `kThemeStatePressed` value. "Window Region Constants" (page 215). Clarified the constant descriptions. `GetNewDialog` (page 268). Specified the creation of a color graphics port under Appearance when the `kDialogFlagsUseThemeBackground` feature bit of the `'dlgx'` resource is set. "Keyboard font character codes" (page 302). Clarified the descriptions of those character codes that differ in right-to-left and left-to-right script systems. "Folder Type Constants" (page 344). Expanded the description of the `kApplicationSupportFolderType` constant. "Glossary" (page 371). Added missing window region definitions.
Dec. 2, 1997	PDF formatting improved.
Nov. 3, 1997	First document release.

APPENDIX

Version History

Glossary

action function A function that performs an action in response to the user holding the mouse button down while the cursor is in a control.

activate event A type of event that indicates that a window is becoming active or inactive. Each activate event specifies the window to be changed and the direction of the change (that is, whether it's becoming active or inactive).

active control A control in which the Control Manager responds to a user's mouse actions by providing visual feedback (for example, by switching a button to its depressed state).

active window The frontmost window on the desktop, the one in which the user is currently working. The active window is designated by racing stripes in the title bar, active controls, and highlighted selections.

A5 world On 68K-based computers, an area of memory in an application's partition that contains the QuickDraw global variables, the application global variables, the application parameters, and the jump table—all of which are accessed through the A5 register.

alert An alert sound, an alert box, or both. Alerts warn the user of an unusual or a potentially undesirable situation occurring within an application. See also **alert box** and **alert sound.**

alert box A window that an application displays on the screen to warn the user or to report an error to the user. An alert box typically consists of text describing the situation and buttons that require the user to acknowledge or rectify the problem. An alert box may or may not be accompanied by an alert sound. See also **caution alert, note alert,** and **stop alert.**

alert color table resource A resource (of type `'actb'`) that lets an application display an alert box using colors other than the system's default window colors.

alert resource A resource (of type `'ALRT'`) that specifies alert sounds, a display rectangle, and an item list for an alert box.

alert sound An audible signal from the Macintosh speaker that warns the user of an unusual or a potentially undesirable situation occurring within an application. An alert sound may or may not be accompanied by an alert box.

alias An object that represents another file, directory, or volume.

alias file A file that contains a record that contains a pointer to another file, directory, or volume. An alias file is displayed by the Finder as an alias.

alias record A data structure created by the Alias Manager to identify a file, directory, or volume.

alias target The file, directory, or volume described by the alias record.

Appearance Manager The operating system service that provides the underlying support for themes. The Appearance Manager manages all aspects of themes and theme switching, including the Appearance control panel, support for a variety of color data, and support for animation and sound.

Appearance-compliant A feature that supports the Appearance Manager.

Apple event A high-level event whose structure and interpretation are determined by the Apple Event Interprocess Messaging Protocol.

Apple Menu Items folder A directory located in the System Folder for storing desk accessories, applications, folders, and aliases that the user wants to display in and access from the Apple menu.

application-defined filter function An application-defined function to handle event filtering. See also **event filter function.**

application heap An area of memory in the application heap zone in which memory is dynamically allocated and released on demand. The heap can contain application code, data structures, resource maps, and other items as needed.

application partition A partition of memory reserved for use by an application. The application partition consists of free space, the application heap, the application's stack, and the application's A5 world.

asynchronous arrow A control which indicates through a simple animation that a background process is in progress. Compare **progress indicator.**

auto-key event An event indicating that a key is still down after a certain amount of time has elapsed.

auxiliary window record A data structure that the Window Manager uses to tie together a list of windows and their corresponding window color information tables.

background process A process that isn't currently interacting with the user. Compare **foreground process.**

bevel button A button containing a self-descriptive icon, picture, text, or any combination of the three, that performs an action when pressed.

bundle bit A flag in a file's Finder information record that informs the Finder that a bundle (`'BNDL'`) resource exists for the file. A file's Finder information record is stored in a volume's catalog file. The Finder uses the information in the bundle resource to associate icons with the file.

catalog file A special file, located on a volume, that contains information about the hierarchical organization of files and folders on that volume.

caution alert An alert box that warns the user of an operation that may have undesirable results if allowed to continue. A caution alert gives the user the choice of continuing the action (by clicking the OK button) or stopping the action (by clicking the Cancel button). A caution alert is

identified by an icon bearing an exclamation point in the upper-left corner of the alert box. See also **note alert** and **stop alert.**

character code A value that represents a particular character. The character code that is generated depends on the virtual key code and the state of the modifier keys. In the Roman script system, character codes are specified in the extended version of ASCII (the American Standard Code for Information Interchange).

checkbox A control that appears onscreen as a small square with an accompanying title. A checkbox displays one of three settings: on (indicated by an X inside the box), off, or mixed (indicated by a dash inside the box. When the user clicks a checkbox, the application reverses its setting. See also **radio button.**

clock control A control that combines the features of little arrows and an edit text field into a control which displays a date and/or time.

close box The small white box on the left side of the title bar of an active window. Clicking it closes the window.

close region The area occupied by a window's close box. See also **close box.**

collapse box A square control which appears on the far right of a window's title bar. Clicking it once hides all of the window except the title bar; clicking it again makes the window reappear.

collapse region The area occupied by a window's collapse box. See also **collapse box.**

Command-key equivalent Refers specifically to a keyboard equivalent that the user invokes by holding down the Command key and pressing another key (other than a modifier key) at the same time. Compare **keyboard equivalent.**

content region The part of a window in which the contents of a document, the size box, and the window controls (including the scroll bars) are displayed.

context The information about a process maintained by the Process Manager. This information includes the current state of the process, the address and size of its partition, its type, its creator, a copy of its low-memory global variables, information about its `'SIZE'` resource, and a process serial number.

contextual menu A pop-up menu containing useful commands and assistance specific to the item being pointed at by the cursor.

control An onscreen object that the user can manipulate with the mouse. By manipulating a control, the user can take an immediate action or change a setting to modify a future action.

control color table In an item color table resource, a specification for the colors used to draw the various parts of a control.

control definition function A function that defines the appearance and behavior of a control. A control definition function, for example, draws the control. See also **standard control definition functions.**

control definition ID A number passed to control-creation functions to indicate the type of control. It consists of the control definition function's resource ID (in the upper 12 bits) and a variation code (in the lower 4 bits).

control list A series of entries pointing to the descriptions of the controls associated with the window.

Control Manager A collection of functions that applications use to create and manipulate controls, especially those in windows.

Control Panels folder A directory located in the System Folder for storing control panels, which allow users to modify the work environment of their Macintosh computer.

control record A data structure of type `ControlRecord`, which the Control Manager uses to store all the information it needs for its operations on a control.

current menu list A data structure that contains handles to the menu records of all menus in the current menu bar and the menu records of any submenus or pop-up menus that an application inserts into the list.

current process The process that is currently executing and whose direct data area or A5 world is valid; this process can be in the background or the foreground.

cursor Any 256-bit image, defined by a 16-by-16 bit square. The mouse driver displays the current cursor and maps the movement of the mouse to relative locations on the screen as the user moves the mouse.

custom alert box An alert box whose upper-left corner contains blank space or displays an icon other than those used by caution alerts, stop alerts, or note alerts.

customized icon An icon created by the user or by an application and stored with a resource ID of –16455 in the resource fork of a file. A file with a customized icon has the `hasCustomIcon` bit set in its Finder flags field.

data fork The part of a file that contains application code (on PowerPC–based computers) and data accessed using the File Manager. Data stored by the File Manager usually corresponds to data entered by the user; the application creating a file can store and interpret the data in the data fork in whatever manner is appropriate.

default button In an alert box or a dialog box, the button whose action is invoked when the user presses the Return key or the Enter key. The Dialog Manager automatically draws a bold outline around the default button in alert boxes; applications should draw a bold outline around the default button in dialog boxes. The default button should invoke the preferred action, which, whenever possible, should be a "safe" action—that is, one that doesn't cause loss of data.

desktop The working environment displayed on the Macintosh computer: the background area on the screen.

desktop database A Finder-maintained database of icons, file types, applications, version data, and comments for all volumes over 2 MB. Compare **Desktop file.**

Desktop file A resource file in which the Finder stores icons, file types, applications, version data, and comments for all volumes less than 2 MB. Compare **desktop database.**

Desktop Folder A directory, located at the root level of each volume, used by the Finder for storing information about the icons that appear on the desktop area of the screen. The Desktop Folder is invisible to the user. What the user sees onscreen is the union of the contents of Desktop Folders for all mounted volumes.

dial A control, similar to a scroll bar, that graphically represents the ranges of values that a user can set or that simply displays the value, magnitude, or position of something, typically in some pseudo-analog form.

dialog box A window that an application displays on the screen to solicit information from the user before the application carries out the user's command. See also **modal dialog box, modeless dialog box,** and **movable modal dialog box.**

dialog color table resource A resource (of type `'dctb'`) that lets an application display a dialog box using colors other than the system's default window colors.

Dialog Manager A collection of functions that applications use to implement alerts and dialog boxes.

dialog record A data structure of type `DialogRecord` that the Dialog Manager uses to create dialog boxes and alerts.

dialog resource A resource (of type `'DLOG'`) that specifies the window type, display rectangle, and item list for a dialog box.

disabled item In an alert box or a dialog box, an item for which the Dialog Manager does not report user events. An example of a disabled item is static text, which typically does not respond to clicks.

disclosure triangle A triangular control governing how items are displayed in a list. The disclosure triangle can point in two directions: right and down. When the disclosure triangle points to the right, one item is displayed in the list. When the arrow points downward, the original item and its subitems are displayed in the list.

disk-inserted event An event indicating that a disk has been inserted into a disk drive.

display rectangle A rectangle that defines the size and location of an item in an alert box or a dialog box. The display rectangle is specified in an item list and uses coordinates local to the alert box or dialog box.

divider A line used in menus to separate groups of menu items.

document window A window in which the user enters text, draws graphics, or otherwise enters or manipulates data.

drag region The window frame, including the title bar and window outline, but excluding the close box, zoom box, and

collapse box. The user can move a window on the desktop by dragging the drag region. See also **frame.**

edit text field A control that appears as a rectangular box inside a dialog box. The user enters text in the edit text field to provide information to an application. Compare **static text field.**

edition The data written to an edition container by a publisher. A publisher writes data to an edition whenever a user saves a document that contains a publisher, and subscribers in other documents may read the data from the edition whenever it is updated.

embedding hierarchy The hierarchy used by the Dialog Manager to indicate the order in which controls are embedded. For example, a tab control may have a radio button control embedded within it.

enabled item In an alert box or a dialog box, an item for which the Dialog Manager reports user events. For example, the Dialog Manager reports clicks in an enabled OK button.

event The means by which the Event Manager communicates information about user actions, changes in the processing status of the application, and other occurrences that require a response from the application.

event filter function An application-defined function that supplements the Dialog Manager's ability to handle events—for example, an event filter function can test for disk-inserted events and can allow background applications to receive update events.

Event Manager The collection of functions that an application can use to receive information about actions performed by the user, to receive notice of changes in the processing status of the application, and to communicate with other applications.

event mask An integer with one bit position for each event type. You specify an event mask as a parameter to Event Manager functions to specify the event types you want your application to receive, thereby disabling (or "masking out") the events you are not interested in receiving.

event record A data structure of type `EventRecord` that your application uses when retrieving information about an event. The Event Manager returns, in an event record, information about what type of event occurred (a mouse click or keypress, for example) and additional information associated with the event.

extended menu resource In Appearance-compliant applications, a resource of type `'xmnu'` that contains additional menu information not contained in the menu resource. For a given menu, both the menu resource and extended menu resource have the same ID. Compare **menu resource.**

Extensions folder A directory located in the System Folder for storing system extension files such as printer and network drivers and files of types `'INIT'`, `'scri'`, and `'appe'`.

file A named, ordered sequence of bytes stored on a Mac OS volume, divided into a data fork and a resource fork.

Finder An application that works with the system software to keep track of files and manage the user's desktop display.

focus ring A colored border that highlights the currently active edit text field or scrolling list in a dialog box in order to indicate to user which item has keyboard focus. See also **keyboard focus.**

folder descriptor A data structure that describes a folder and its contents. Folders described by a folder descriptor can be found using the `findFolder` function, even if nested inside another folder.

Fonts folder A directory located in the System Folder for storing fonts.

foreground process The process currently interacting with the user; it appears to the user as the active application. The foreground process displays its menu bar, and its windows are in front of the windows of other applications. Compare **background process.**

frame The part of a window drawn automatically by the Window Manager, namely, the title bar, including the close box, zoom box, and collapse box, and the window's outline.

global coordinate system The coordinate system that represents all potential QuickDraw drawing space. The origin of the global coordinate system—that is, the point (0,0)—is at the upper-left corner of the main screen. Compare **local coordinate system.**

graphics port A complete, individual drawing environment with an independent coordinate system. Each window is drawn in a graphics port.

gray area The area within a scroll bar, excluding the scroll arrows and the scroll box. When the user clicks the gray area of a scroll bar, the application moves the displayed area of the document by an entire window less one line (or column, row, or character).

gray region A region that represents all available desktop area—that is, a collection of rounded-corner rectangles representing the display areas of all monitors available to a computer.

group box A control that consists of a rectangular frame which may or may not contain a title. It is used to provide a well-defined area in a dialog box into which text, pictures, icons or other controls can be embedded.

grow image An outline of a window's new frame, drawn on the screen while the user is resizing the window with the size box.

help balloon A rounded-rectangle window that contains explanatory information for the user. With tips pointing at the objects they annotate, help balloons look like bubbles used for dialog in comic strips. Help balloons are turned on by the user from the Help menu; when Balloon Help assistance is on, a help balloon appears whenever the user moves the cursor over an area that is associated with it.

hierarchical menu A menu to which a submenu is attached.

high-level event An event sent from one application to another requesting transfer of information or performance of some action.

high-level event queue A separate queue that the Event Manager maintains to store high-level events transmitted to an application. The Event Manager maintains a high-level event queue for each open application capable of receiving high-level events.

hot spot A point that the mouse driver uses to align the cursor with the mouse location.

human interface objects Execution-time structures that encapsulate one or more human interface elements, such as windows, dialog boxes, or controls. Human interface objects support such object-oriented programming features as inheritance, subclassing, and polymorphism.

icon An image that represents an object, a concept, or a message.

icon family The set of icons that represent an object—such as an application or a document—displayed by the Finder. An entire icon family consists of large (32-by-32 pixel) and small (16-by-16 pixel) icons, each with a mask, and each available in three different versions of color: black and white, 4 bits of color data per pixel, and 8 bits of color data per pixel.

icon suite One or more handles to icon data that represents icons from a single icon family. Some Icon Utilities functions accept a handle to an icon suite and draw the appropriate icon from that suite for the destination rectangle and the bit depth of the display device.

image well A control that is used to display non-text visual content on a white background surrounded by a rectangular frame.

inactive control A control that has no meaning or effect in the current context—for example, the scroll bars in an empty window. The Control Manager dims inactive controls or otherwise visually indicates their inactive state.

inactive window A window in which the user is not working.

indicator A moving part in a dial or slider control. A user moves an indicator to set a value, and an application moves it to indicate the current setting of the control. In a scroll bar, the scroll box is the indicator.

item color table resource A resource (of type `'ictb'`) that an application can use to display an alert box or a dialog box with items using a typeface, font style, font size, or colors other than the system's default font and colors. (For an application to use a nonstandard typeface, font style, or font size, the user must have a color monitor.)

item list A resource (of type `'DITL'`) that specifies the items—such as buttons and static text—to display in an alert box or a dialog box.

item number An integer that identifies an item in either a menu or a dialog box. Menu items are assigned item numbers starting with 1 for the first menu item in the menu, 2 for the second menu item in the menu, and

so on, up to the number of the last menu item in the menu. Dialog items are assigned numbers that correspond to the item's position in its item list. For example, the first item listed in a dialog item list is item number 1.

keyboard equivalent A keyboard combination of one or more modifier keys and a character key that invokes a corresponding menu command when pressed by the user.

keyboard focus A property that determines which control in a dialog will receive all keystrokes (keyboard events), as selected by keyboard navigation or clicking.

key-down event An event indicating that the user pressed a key on the keyboard.

key-up event An event indicating that the user released a key on the keyboard.

list box A control that combines a rectangular frame, scroll bar(s), and a scrolling list.

little arrow Up- and down-arrows accompanying a text box that contains a value, such as a date. Clicking the up arrow increases the value displayed. Clicking the down arrow decreases the value displayed.

local coordinate system The coordinate system defined by the port rectangle of a graphics port. When the Window Manager creates a window, it places the origin of the local coordinate system at the upper-left corner of the window's port rectangle. Compare **global coordinate system.**

location name An identifier for the network location of the computer on which a Program-to-Program Communications (PPC) port resides. A location name consists of an object string, a type string, and a zone.

low-level event The type of event returned by the Event Manager to report very low level hardware and software occurrences. Low-level events report actions by the user, changes in windows on the screen, and that the Event Manager has no other events to report. Compare **high-level event, operating-system event.**

major switch A change of the foreground process. The Process Manager switches the context of the foreground process with the context of a background process (including the A5 worlds and low-memory global variables) and brings the background process to the front, sending the previous foreground process to the background. See also **context.**

menu A user interface element you can use in your application to allow the user to view or choose an item from a list of choices and commands that your application provides. See also **hierarchical menu, pop-up menu, pull-down menu,** and **submenu.**

menu bar A white rectangle that is tall enough to display menu titles in the height of the system font and system font size, and with a black lower border that is one pixel tall. The menu bar extends across the top of the startup screen and contains the title of each available pull-down menu.

menu bar definition function A function that draws the menu bar and performs most of the drawing activities related to the display of menus when the user moves the cursor between menus. This function, in conjunction with the menu definition function, defines the general appearance and behavior of menus.

menu bar entry A menu color entry record that contains 0 in both the `mctID` and `mctItem` fields. A menu bar entry defines the color for an application's menu bar and defines default colors for its menu titles, menu items, and background color of menus.

menu bar resource A resource (of type `'MBAR'`) that specifies the order and resource ID of each menu in a menu bar.

menu color entry record A data structure of type `MCEntry` that defines the colors for an application's menu bar, menus, or menu items. The first two fields of a menu color entry record, `mctID` and `mctItem`, define whether the entry is a menu bar entry, a menu title entry, or a menu item entry.

menu color information table An array of menu color entry records, maintained by the Menu Manager, that define the standard color for the menu bar, titles of menus, text and characteristics of menu items, and background color of a displayed menu. If you do not add any entries to this table, the Menu Manager draws your menus using the default colors, black on white.

menu color information table resource A resource (of type `'mctb'`) that specifies the colors for an application's menu bar, menus, and menu items.

menu definition function A function that performs all the drawing of menu items within a specific menu. This function, in conjunction with the menu bar definition function, defines the general appearance and behavior of menus.

menu ID A number that you assign to a menu in your application. Each menu in your application must have a unique menu ID.

menu item In a menu, a rectangle with text and other characteristics identifying a command that the user can choose.

menu item entry A menu color entry record that contains nonzero values in both the `mctID` and `mctItem` fields. A menu item entry defines colors for the mark, text, and keyboard equivalent of items in a specific menu. It also defines the default background color of a menu.

menu title entry A menu color entry record that contains nonzero values in both the `mctID` and `mctItem` fields. A menu item entry defines colors for the mark, text, and keyboard equivalent of items in a specific menu. It also defines the default background color of a menu.

menu list A data structure that contains handles to the menu records of one or more menus (although a menu list can be empty). Compare **current menu list.**

Menu Manager The collection of functions that an application can use to create, display, and manage its menus.

menu record A data structure of type `MenuInfo` that the Menu Manager uses to maintain information about a menu.

menu resource A resource (of type `'MENU'`) that specifies the menu title and the individual characteristics of items in a menu.

menu title entry A menu color entry record that contains a nonzero value in the `mctID` field and contains 0 in the `mctItem` field. A menu title entry defines colors for the title, items, and background color of a specific menu. It also defines the default menu bar color.

minimum partition size The actual partition size limit below which an application cannot run.

minor switch A change in the context of a process. The Process Manager switches the context of a process to give time to a background process without bringing the background process to the front.

modal dialog box A dialog box that puts the user in the state or "mode" of being able to work only inside the dialog box. A modal dialog box resembles an alert box. The user cannot move a modal dialog box and can dismiss it only by clicking its buttons. See also **modeless dialog box** and **movable modal dialog box.**

modeless dialog box A dialog box that looks like a document window without a collapse box or scroll bars. The user can move a modeless dialog box, make it inactive and active again, and close it like any document window. See also **modal dialog box** and **movable modal dialog box.**

modifier keys The Shift, Option, Command, Control, and Caps Lock keys. You can use modifier keys in conjunction with character keys to form keyboard equivalents. See also **keyboard equivalents.**

mouse-down event An event indicating that the user pressed the mouse button.

mouse location The location of the cursor at the time the event occurred.

mouse-moved event An event indicating that the cursor is outside of a specified region.

mouse-up event An event indicating that the user released the mouse button.

movable modal dialog box A modal dialog box that has a title bar (with no close box) by which the user can drag the dialog box. See also **dialog box, modal dialog box,** and **modeless dialog box.**

note alert An alert box that informs users of a minor mistake that won't have any disastrous consequences if left as is. Usually a note alert simply offers information, and the user responds by clicking the OK button. A note alert is identified by an icon bearing a face and a cartoonlike dialog balloon in the upper-left corner of the alert box. See also **caution alert** and **stop alert.**

null event An event indicating that no events of the requested types exist in the application's event stream.

offset point The point in a region whose horizontal and vertical offsets from the upper-left corner of the region's enclosing rectangle are the same as the offsets of a specified point. The `DragGrayRgn` function

uses an offset point to limit the motion of a region and to calculate the distance a region has moved.

operating-system event An event returned by the Event Manager to communicate information about changes in the operating status of applications (suspend and resume events) and to report that the user has moved the cursor outside of an area specified by the application (mouse- moved events). Compare **low-level event, high-level event.**

Operating System Event Manager The collection of low-level functions that manage the Operating System event queue.

Operating System event queue A queue that the Operating System Event Manager creates and maintains. The Operating System Event Manager detects and reports low-level hardware-related events such as mouse clicks, keypresses, and disk insertions and places these events in the Operating System event queue.

part code An integer from 1 through 253 that stands for a particular part of a control. The `FindControl` and `TrackControl` functions return a part code to indicate the location of the cursor when the user presses the mouse button.

placard A rectangular control used as an information display.

pop-up menu A menu that appears elsewhere than the menu bar. The Control Manager provides a control definition function for applications to use when implementing pop-up menus.

pop-up button A button which, when the user presses the mouse and the cursor is over the pop-up button, displays associated menu items.

port name A unique identifier for a particular application on a computer, used for the purposes of communication between applications. A port name consists of a name string, a type string, and a script code.

port rectangle An entry in the graphics port data structure, described in *Inside Macintosh: Imaging With QuickDraw.* Ordinarily, the port rectangle represents the area of a graphics port available for drawing—that is, the content region of a window.

Preferences folder A directory located in the System Folder for holding files that record users' configuration settings for applications on a particular Macintosh computer.

preferred partition size The partition size at which an application can run most effectively. The Operating System attempts to secure this partition size upon launch of the application.

PrintMonitor Documents folder A directory located in the System Folder for storing spooled documents waiting to be printed.

process An open application or, in some cases, an open driver. (Only drivers that are not opened in the context of another application are considered processes.)

process serial number A number assigned by the Process Manager to identify a particular instance of an application during a single boot of the local machine.

progress indicator A control indicating that a lengthy operation is occurring. Two types of progress indicators can be used: an *indeterminate progress indicator* reveals that an operation is occurring but does not indicate its duration; a *determinate progress indicator* displays how much of the operation has been completed.

pull-down menu A menu that is identified by a menu title (a word or an icon) in the menu bar.

push button A control that appears on the screen as a rounded rectangle with a title centered inside. When the user clicks a push button, the application performs the action described by the button's title. Button actions are usually performed instantaneously. Examples include completing operations defined by a dialog box and acknowledging an error message in an alert box.

query document A file of file type `'qery'` containing commands and data in a format appropriate for a database or other data source. An application uses high-level Data Access Manager functions to open a query document.

radio button A control that appears onscreen as a small circle. A radio button displays one of two settings: on (indicated by a black dot inside the circle) or off. A radio button is always a part of a group of related radio buttons in which only one button can be on at a time. When the user clicks an unmarked radio button, the application turns that button on and turns the other buttons in its group off.

Rescued Items from *volume name* folder A directory located in the Trash directory and created by the Finder at system startup, restart, or shutdown only when it finds items in the Temporary Items folder, usually after a system crash. The Rescued Items from *volume name* folder is named for the volume on which the Temporary Items folder exists. When a user empties the Trash, all Rescued Items folders disappear.

resource Any data stored according to a defined structure in a resource fork of a file; the data in a resource is interpreted according to its resource type.

resource fork The part of a file that contains the file's resources. A resource fork consists of a resource map and resources.

resource ID A number that identifies a specific resource of a given resource type.

resource type A sequence of four characters that uniquely identifies a specific type of resource.

resume event An event indicating that an application has been switched back into the foreground and can resume interacting with the user.

return receipt A high-level event that indicates whether the other application accepted the high-level event sent to it by your application.

root control The base control in a window hierarchy. To embed controls in a window, you must create a root control for that

window. The root control is the container for all other window controls. Once you have created a root control, newly-created controls will automatically be embedded in the root control when you call other functions.

scroll arrow An arrow at either end of a scroll bar. When the user clicks a scroll arrow, the application moves a document or list one line (or some similar measure) in the direction of the arrow. When the user holds the mouse button down while the cursor is over a scroll arrow, the application moves the document or list continuously in the direction of the arrow.

scroll bar A control with which the user can change the portion of a document displayed within a window. A scroll bar is a light gray rectangle with scroll arrows at each end. Windows can have a horizontal scroll bar, a vertical scroll bar, or both. A vertical scroll bar lies along the right side of a window. A horizontal scroll bar runs along the bottom of a window. Inside the scroll bar is a rectangle called the scroll box. The rest of the scroll bar is called the gray area. The user can move through a document by manipulating the parts of the scroll bar.

scroll box A box that slides up and down or back and forth across a scroll bar. The position of the scroll box in a scroll bar indicates the position of the window contents relative to the entire document. When the user drags the scroll box, the application displays a different portion of the document.

script code A value that defines the script to use when displaying text. Scripts (more often called text encodings) generally reflect differences in characters or languages based on geographical location. Script codes are sometimes called *script IDs.* See also **text encoding, text encoding specification.**

signature A resource whose type is defined by a four-character sequence that uniquely identifies an application to the Finder. A signature is located in an application's resource fork.

size box A box in the lower-right corner of windows that can be resized. Dragging the size box resizes the window.

size region The area occupied by a window's size box. See also **size box.**

size resource A resource (of type `'SIZE'`) that specifies the operating characteristics, minimum partition size, and preferred partition size of an application.

slider A control that displays a range of values, magnitudes, or positions. A horizontally- and vertically-mobile indicator is used to increase or decrease the value.

standard control definition functions Three control definition functions, stored as `'CDEF'` resources in the System file. The `'CDEF'` resource with resource ID 0 defines the look and behavior of buttons, checkboxes, and radio buttons; the `'CDEF'` resource with resource ID 1 defines the look and behavior of scroll bars; and the `'CDEF'` resource with resource ID 63 defines the look and behavior of pop-up menus.

standard state The size and location that an application deems the most convenient for a window.

Startup Items folder A directory located in the System Folder for storing applications and desk accessories that the user wants started up every time the Finder starts up.

static text field A control that displays static (unchangeable by the user) text labels in a window. Compare **edit text field.**

stationery pad A document that a user creates to serve as a template for other documents. The Finder tags a document as a stationery pad by setting the `isStationery` bit in the Finder flags field of the file's file information record. An application that is asked to open a stationery pad should copy the template's contents into a new document and open the document in an untitled window.

stop alert An alert box that informs the user of a problem or situation so serious that the user's desired 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. A stop alert is identified by an icon of an upraised hand in the upper-left corner of the alert box. See also **caution alert** and **note alert.**

structure region The entire area occupied by a window, including both the window frame and the content region.

submenu A menu that is attached to another menu.

suspend event An event indicating that the execution of your application is about to be suspended as the result of a major switch. The application is suspended at the application's next call to `WaitNextEvent` or `EventAvail`.

system alert sound A sound resource that is stored in the System file and played whenever system software or an application uses the Sound Manager function `SysBeep`. With the Sound control panel, the user can select which sound to use.

System file A file, located in the System Folder, that contains the basic system software plus some system resources, such as sound and keyboard resources.The System file behaves like a folder in this regard: although it looks like a suitcase icon, double-clicking it opens a window that reveals movable resource files (such as sounds, keyboard layouts, and script system resource collections) stored in the System file.

System Folder A directory containing the software that Macintosh computers use to start up. The System Folder includes a set of folders for storing related files, such as preferences files that an application might need when starting up.

tab control A control that appears as a row of folder tabs on top of a pane. It allows multiple panes to appear in the same window. See also **pane**.

Temporary Items folder A directory located at the root level of a volume for storing temporary buffer files created by applications. The Temporary Items folder is invisible to the user.

text encoding The correspondence between numerical character codes and the final printable glyphs. For instance, 0x42 is the ASCII code for the letter B. Text encodings can describe arbitrary character sets (such as ASCII or Unicode) or those based on geographical differences (for example, Japanese Kanji).

text encoding specification A value of type `TextEncoding` that defines a text encoding.

text style table In an item color table resource, a specification for the typeface, font style, font size, and color of text in an editable text item or a static text item.

title bar The bar at the top of a window that displays the window name, contains the close and zoom boxes, and indicates whether the window is active.

title bar region The entire area occupied by a window's title bar, including the title text region. See also **title bar** and **title text region.**

title text region That portion of a window's title bar that is occupied by the name of the window. See also **title bar.**

Toolbox Event Manager See **Event Manager.**

Trash folder A directory at the root level of a volume for storing files that the user has moved to the Trash icon. After opening the Trash icon, the user sees the collection of all items that the user has moved to the Trash icon—that is, the union of appropriate Trash directories from all mounted volumes. A Macintosh computer set up to share files among users in a network environment maintains separate Trash subdirectories for remote users within its shared Trash directory. The Finder empties a Trash directory (or, in the case of a file server, a Trash subdirectory) only when the user of that directory chooses the Empty Trash command.

universal procedure pointer A generalized procedure pointer that is used when there is some ambiguity about the calling conventions of the code being called. As used by the Mixed Mode Manager, a universal procedure pointer can be either a 68K procedure pointer or the address of a routine descriptor.

update event An event indicating that the contents of a window need updating.

update region A region maintained by the Window Manager that includes the parts of a window's content region that need updating. The Event Manager generates update events as necessary, based on the contents of the update region, telling your application to update a window.

user state The size and location that the user has established for a window.

utility window A type of box that has some but not all features of a regular window. A utility window has a bar at the top by which it can be dragged and a close box, but it does not necessarily have a title, and is nonscrolling. Utility windows typically float above all other windows in your application.

variation code A number that selects among variations supported by a single window definition function or control definition function. The variation code is

stored in the low-order 4 bits of the window definition ID or control definition ID. See also **control definition function, control definition ID, window definition function,** and **window definition ID.**

virtual key code A value that represents the key pressed or released by the user; this value is always the same for a specific physical key on a keyboard. Compare **character code.**

visible region The part of a window's graphics port that's actually visible on the screen—that is, the part that's not covered by other windows.

visual separator A panel displaying horizontal, vertical, or rectangular elements used to visually separate other panels in a window. Rectangular visual separators can contain titles.

window An area on the screen that displays information, including user documents as well as communications such as alert boxes and dialog boxes. The user can open or close a window; move it around on the desktop; and sometimes change its size, scroll through it, and edit its contents.

window color table The data structure in which the Window Manager stores the colors to be used for drawing a window's frame and for highlighting selected text.

window definition function A function that defines the general appearance and behavior of a window. The Window Manager calls the window definition function to draw the window's frame, determine what region of the window the cursor is in, draw the window's size box, draw the window's zoom box, move and resize the window, and calculate the window's structure and content regions.

window definition ID An integer that specifies the resource ID of a window definition function in the upper 12 bits and an optional variation code in the lower 4 bits. When creating a new window, your application supplies a window definition ID either as a field in the `'WIND'` resource or as a parameter to the `NewWindow` or `NewCWindow` function.

window header A control that runs along the top of a window's content region and provides information about the window's contents.

window list A list maintained by the Window Manager of all windows on the desktop. The frontmost window is first in the window list, and the remaining windows appear in the order in which they are layered on the desktop.

Window Manager port A graphics port that represents the desktop area on the main monitor—that is, a rounded-corner rectangle that occupies all of the main monitor except for the area occupied by the title bar.

window origin The upper-left corner of a window. Usually specified as (0,0), the window origin is expressed in coordinates local to the window.

window record A data structure of type `WindowRecord` (or `CWindowRecord`) in which the Window Manager stores a window's characteristics, including the window's graphics port, title, visibility status, and control list.

window region Special-purpose region of a window. See also **close region, collapse region, content region, drag region, size region, structure region, title bar region, title text region,** and **zoom region.**

window type A collection of characteristics—such as the shape of the window's frame and the features of its title bar—that describe a window.

zoom box A box in the right side of a window's title bar that the user can click to alternate between two different window sizes (the user state and the standard state).

zoom region The area occupied by a window's zoom box. See also **zoom box.**

Index

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W

X

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITERS
Lisa Karpinski, Donna S. Lee, Judith Rosado, and Jun Suzuki

ILLUSTRATORS
David Arrigoni and Karin Stroud

PRODUCTION EDITOR
Glen Frank

PROJECT MANAGER
Tony Francis

Special thanks to Ed Voas and Matt Ackeret

Acknowledgments to Eric Anderson, Guy Fullerton, Pete Gontier, C.K. Haun, Eric Koebler, Dave Lyons, Patrick McClaughry, Matt Mora, Greg Robbins, Jeff Shulman, and Chris Thomas