INSIDE MACINTOSH

Human Interface Toolbox

WWDC Release

May 1996

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

Simultaneously published in the United States and Canada.

Contents

Introduction to the Mac OS 8 Toolbox

Contents

The Mac OS 8 Toolbox consists of system software services that you can use to create your application's human interface elements and present them to users. The Toolbox also simplifies a variety of human interface programming tasks and provides low-level support for active assistance.

This chapter introduces some of the standard human interface features of Mac OS 8 applications and the Toolbox services you use to implement them. It also discusses the role of Apple events in Mac OS 8 programming. For more information about Apple events and related Mac OS 8 capabilities, see the accompanying document *Apple Events in Mac OS 8.*

▲ **WARNING**
This document is preliminary and incomplete. All information presented here is subject to change. ▲

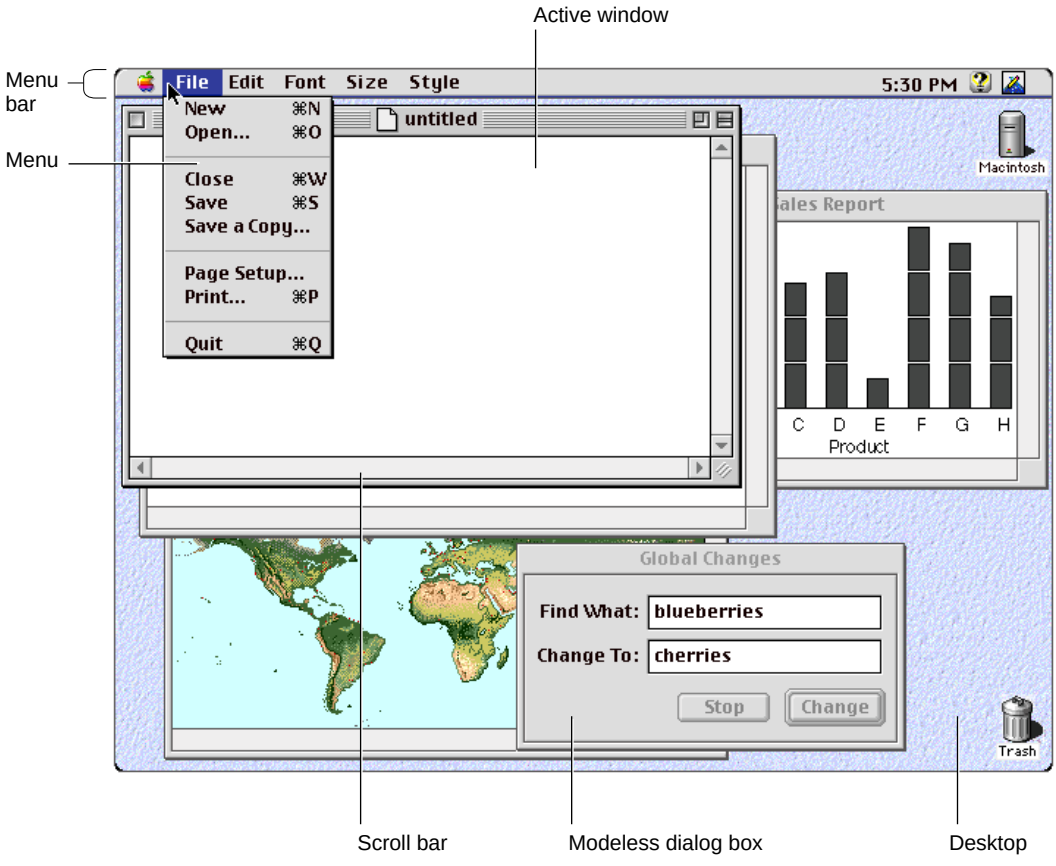
A typical Mac OS 8 application presents users with a carefully designed human interface that allows them to perform actions and accomplish goals according to their own priorities. To ensure that human interface elements share consistent behavior and appearance across all applications, the Mac OS 8 Toolbox provides a comprehensive set of standard interface elements that you can piece together according to your application's needs. This ensures that common elements such as pop-up buttons and sliders work the same way in different applications and coordinate with the appearance of other elements.

The Mac OS 8 Toolbox also supports customization by each user in ways that maintain the overall look and feel of the human interface for that user. For example, users can choose among different themes, or styles—that is, coordinated sets of human interface designs that determine the appearance of human interface elements on a systemwide basis, across multiple applications.

The figures that follow show some of the standard human interface elements provided by the Toolbox and the way their appearance changes when the user switches themes. Figure 1-1 shows how the screen might appear when a user has selected the Apple Default theme and is interacting with a typical Mac OS 8 application, called SurfWriter, that permits simple text editing.

Note
Unless otherwise indicated, the human interface elements illustrated in this chapter use the Apple Default theme. ◆

Figure 1-1 The SurfWriter application as it appears in the Apple Default theme



A user can directly control the SurfWriter application by means of a variety of human interface elements, including

- menus that allow the user to choose commands
- windows that allow the user to enter and edit information
- scroll bars and other controls that the user can manipulate
- dialog boxes that solicit information from the user

Figure 1-2 The SurfWriter application as it appears in an alternate theme

In addition to interacting freely with the application's human interface elements, the user can change the appearance of all windows, controls, menus, and other elements displayed on a single computer by choosing a different theme. Figure 1-2 shows the SurfWriter application as it appears in an alternate theme.

Apple supplies several standard themes. The Apple Default theme shown in Figure 1-1 is built into the system. Users can install additional themes, switch the current theme from one installed theme to another, or remove installed themes whenever they wish.

Overview of the Mac OS 8 Toolbox

The Mac OS 8 Toolbox provides a complete programming model for creating an application's human interface. From the user's point of view, a typical Mac OS 8 application can, among other things,

- respond to input from a keyboard, mouse, or other input device
- display windows, alert boxes, and dialog boxes that present data and various choices about manipulating the data to the user
- display controls that let the user perform actions or manipulate application settings directly with a variety of input devices
- display menus that let the user choose from lists of choices or commands

In general, the user should always be free to choose the next action to perform. To support this freedom in your application, you use the Apple Event Manager, which provides a systemwide mechanism for distributing events in a preemptively safe manner. Events that use this mechanism are called **Apple events.**

To create your application's windows, dialog boxes, controls, and menus, you use **human interface objects (HI objects).** An HI object is a SOM™ object that encapsulates one or more human interface elements. The HI Objects class library, which is part of the Toolbox, provides a unified, object-oriented interface for implementing navigation, mouse interaction, copy, paste, drag and drop, and other standard behavior for HI objects. The library includes definitions for a variety of standard windows, menus, lists, dialog boxes and alert boxes, scroll bars and other controls, editable text boxes, and other human interface elements. You use these standard HI objects to assemble your application's human interface. If necessary, you can also define your own custom HI objects using object-oriented programming (OOP) techniques.

None of the standard HI objects require a traditional Toolbox manager. Instead of calling a manager's functions, you create and manipulate HI objects by calling their methods. Each HI object knows how to draw itself appropriately depending on its state; for example, checkboxes can check and uncheck themselves, sliders can change their appearance in response to dragging, menus can highlight correctly when selected, and so on.

The Toolbox provides default event handlers that route mouse events, window events, text events, and other standard Apple events to the appropriate HI

objects automatically. You can override the default handlers and add handlers for additional events to implement your application's unique behavior.

HI objects commonly make use of another kind of SOM object called an imaging object. An **imaging object** is a drawing engine that can draw a particular type of image data, such as icons, text, pictures, patterns, and so on. Each type of data must be drawn by its corresponding imaging object, but all imaging objects share the same programming interface. HHI objects that need to draw titles, menu items, and list items use imaging objects to do so, so you can use any kind of image data in such elements.

You can store descriptions of HI objects, images, sounds, and other localizable data in resources managed by the Resource Manager. Resources are completely separate from your application's code and thus greatly simplify localization. To read and write resources, you typically use high-level system services such as the HI Objects class library, which in turn call the Resource Manager.

In addition to providing an integrated human interface composed of HI objects, a Mac OS 8 application

- supports copy, paste, and drag and drop
- has characteristic icons that represent the application file and the application's documents in the Finder
- lets users specify application-specific preferences

The following Toolbox services allow you to implement these capabilities:

- The Clipboard Manager and Drag Manager support a consistent user experience for copy, paste, and drag-and-drop operations.
- The Scrap Manager provides a generic transport package format used by the Clipboard Manager and Drag Manager.
- The Finder interface lets you specify icons that represent your application and its documents in the Finder.
- The Preferences Manager controls all application preferences.

When you use the standard HI objects, your application automatically supports themes. The Appearance Manager provides the underlying support for all aspects of themes and theme switching. Because you can combine the standard HI objects to create a wide range of complex human interfaces, you don't often need to use the Appearance Manager directly. If you do need to customize HI objects, you can use Appearance Manager primitives to coordinate the appearance of your objects with the current theme.

Apple Events and the Toolbox

System 7 and earlier versions of the Mac OS require applications to have an event loop, a piece of code that continually polls the system for events and responds to those events appropriately. Although this arrangement allows the user considerable freedom in choosing when to perform various actions, it has limitations in the Mac OS 8 preemptive multitasking environment. The Mac OS 8 event model, based on the Apple events mechanism introduced in System 7, provides a unified interface for events throughout the system, avoids the problems created by polling, and enhances responsiveness to user actions.

From a user's point of view, the way Mac OS 8 handles events is similar to the way the Mac OS has always handled them. For example, the user can type text in a window, select a graphic and copy it, open a new document in a different application, paste in the graphic, open another document, then go back to the first window to select text and change its size, style, or font.

From a programmer's point of view, the Mac OS 8 event model differs significantly from the event loop model. The essence of Mac OS 8 event handling is simple. When your application launches, it expresses an interest in receiving certain events. Your application then informs the Apple Event Manager that it is ready to receive, and the Apple Event Manager blocks the calling task until an event in which your application has expressed an interest arrives. This arrangement takes maximum advantage of priority-based preemptive scheduling, allowing other applications and tasks to receive processing time when your application doesn't need it.

Mac OS 8 includes a set of low-level services, called User Input Services, that receive input from various kinds of input devices, convert the input (for example, mouse actions or key presses) into Apple events, and send the Apple events to an Apple event dispatcher associated with a process. An **Apple event dispatcher** then dispatches the events within a process. Every process has a default dispatcher that provides the initial dispatching for all incoming events, and your application may create additional dispatchers if necessary. To identify which events it's interested in receiving, your application associates event handlers with the default dispatcher using a mechanism analogous to the way System 7 applications install Apple event handlers.

Handlers associated with a process dispatcher typically determine which window within the process an event is directed toward and forward it to that

window. Every window has a single Apple event dispatcher associated with it that provides the initial processing for all events received by that window. Handlers associated with the window dispatcher typically call methods of one or more HI objects within the window for further event processing as needed.

Toolbox Event Routing

The Toolbox defines standard events, including mouse events and window events, for which it also provides default handlers and for which the HI Objects class library defines methods. The Toolbox also provides default handlers for other events, such as key events and text events, generated by other parts of the system. The Toolbox handlers automatically route most events from the default process dispatcher to the target HI objects, which provide methods that can handle specific events. For example, the default handlers route text events to the HI object that currently has **user input focus**—that is, it is the focal point onscreen for user input, whether from a keyboard, a speech input device, or other input devices.

By default, the target HI objects respond automatically to user actions that change an object's state in some standard way, for example by zooming a window to the appropriate monitor, highlighting menu items as the user drags the pointer through them, changing the highlighting of radio buttons, and so on. You don't need to subclass from the standard objects to associate application-specific behavior with state changes in a particular HI object. Instead, you install a callback function after you instantiate the object.

The ability of the standard HI objects to respond to user interaction appropriately is not limited to low-level events such as mouse events and key events. Mac OS 8 supports higher-level events that correspond to interface abstractions such as "select object." The default Toolbox event handlers translate mouse events, key events, speech events, and other events generated by input devices into higher-level events that your application handles the same way regardless of the originating device.

You may selectively override the default event routing by installing your own handlers in the appropriate dispatcher or by defining your own `HIObject` subclass and overriding the appropriate methods. Thus, although the Toolbox provides extensive default event-routing and event-handling capabilities from the level of the process dispatcher to the level of an individual HI object, you can modify the default behavior to any extent necessary at any point along the way. This provides benefits both for application developers, who can take advantage of the default event handling without sacrificing flexibility, and for

framework developers, who can build the framework's event handling on top of the default behavior.

For an introduction to the default routing provided by the Toolbox, see "Toolbox Event Routing" (page 2-3). For definitions of the standard Toolbox events and their default handlers, see "Toolbox Events Reference" (page 3-3).

Periodic and Background Processing

Another difference between the Mac OS 8 event model and a traditional event loop involves **periodic processing,** which is processing that takes place at specified intervals. For example, if the user isn't doing anything else, an application should be able to perform repetitive tasks such as making the caret blink in the active window. To support this kind of processing in Mac OS 8, you use periodic Apple events.

Periodic processing is different from **background processing,** which takes place in the background while the user continues to work. You perform background processing by creating an additional task that executes preemptively and concurrently while the main task that controls the human interface continues to respond to user actions.

For example, it may be desirable for a graphics application to perform intensive calculations related to image processing in the background, allowing the user to continue to interact with the application's human interface without loss of responsiveness. When the calculations are complete, the additional task can transfer the result of the calculations to the main task, which actually draws the image to the screen.

Supporting the Mac OS 8 Event Model

The best way to support the Mac OS 8 event model is to separate the code that controls your application's user interface from the code that responds to the user's manipulation of the interface. This is called **factoring** your application. A fully factored application translates user actions into Apple events that the application sends to itself to handle those actions appropriately. Factoring not only supports the Mac OS 8 event model, but also allows your application to be controlled by means of any scripting language, such as AppleScript, that's based on the Open Scripting Architecture (OSA).

For more information about supporting the Mac OS 8 event model, see the accompanying document *Apple Events in Mac OS 8.*

Human Interface Objects

The HI Objects class library provides an object-oriented interface for human interface elements commonly used by applications. These standard elements are based on SOM classes that all inherit from the abstract superclass `HIObject`, as shown in Figure 1-3.

Figure 1-3 shows the **inheritance hierarchy** for the standard HI object classes—that is, the ways in which classes inherit their methods and other characteristics from other classes. This inheritance hierarchy has no relationship to the containment hierarchies for the runtime objects you instantiate from the HI object classes.

A **containment hierarchy** describes which instantiated HI objects are contained within which other HI objects. For example, a radio button panel can be located inside a radio button group panel, which is inside a dialog box, which is inside a window.

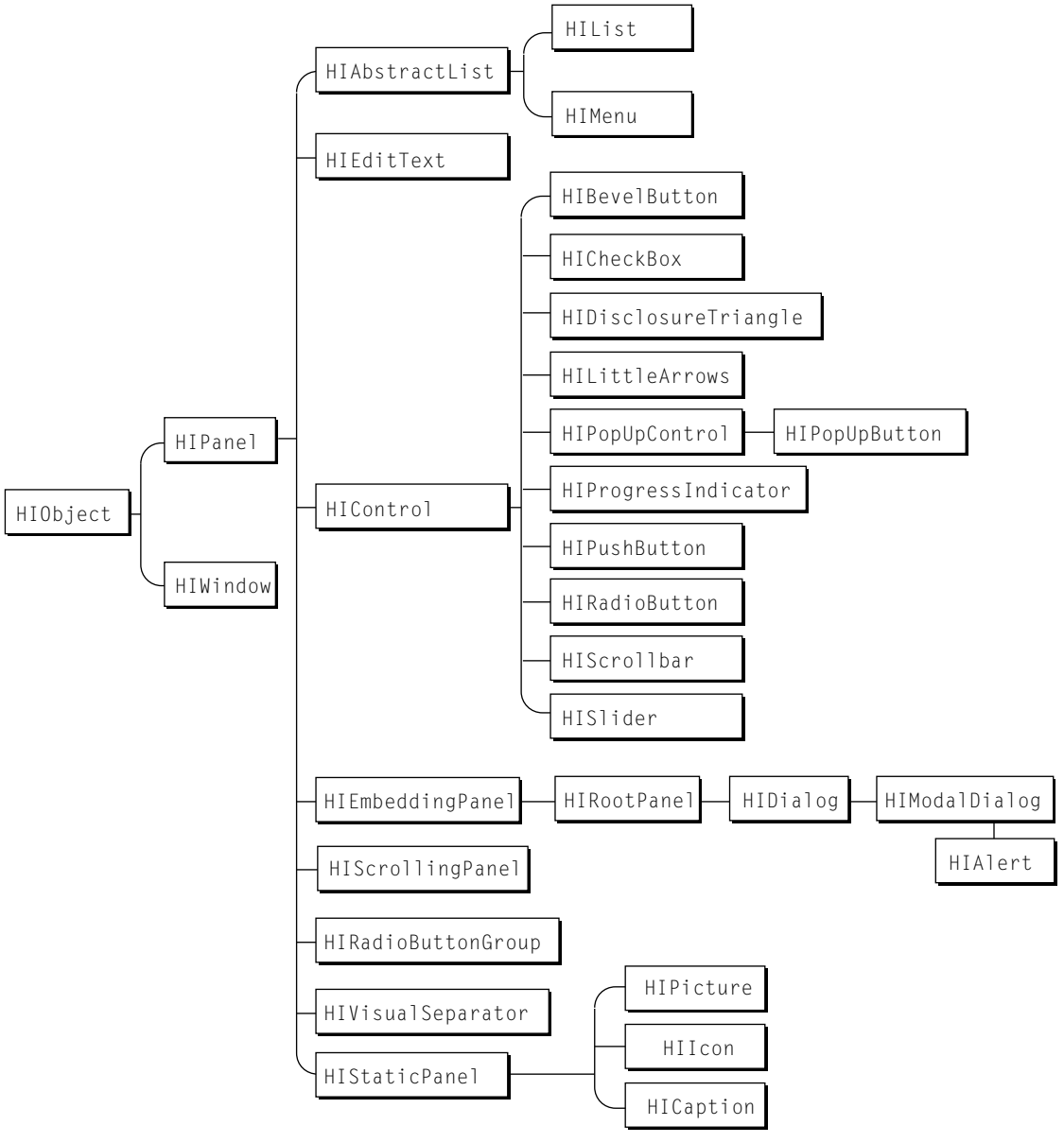
Class `HIObject` has two direct subclasses, `HIWindow` and `HIPanel`. A **window** is a container for all other kinds of HI objects, including menus and dialog boxes. A **panel** is any HI object that can be placed in a window. All HI objects other than windows are panels.

Embedding panels and root panels play an important role in every application's human interface. An **embedding panel** (class `HIEmbeddingPanel`) is a special kind of panel that can contain other panels. Embedding panels are useful for assembling compound panels from the standard panels.

A **root panel** (class `HIRootPanel`) is an embedding panel that fills a window's content area and to which the window passes all events that affect the window's content. The root panel in turn passes events to other panels that it contains. For example, a modal dialog box is a specialized root panel that tracks user interaction with the panels it contains and takes care of all event handling required to enforce its modal state.

Note
HI objects can be used within OpenDoc parts, but they aren't intended to be as large or as powerful as parts. Instead, they facilitate the assembly of an integrated human interface from smaller, simpler elements. ◆

Figure 1-3 The inheritance hierarchy for the HI Objects class library .



The classes shown in Figure 1-3 play the following roles:

- `HIObject` is the abstract superclass for all HI object classes. Its methods perform operations such as handling events, manipulating an object's location, enabling and disabling it, setting its visibility, controlling user input focus, and imaging.
- `HIWindow` is one of the two direct subclasses of `HIObject`. It is a concrete class that defines the standard Mac OS 8 windows. Its methods perform operations on a single window, including handling events within the window, highlighting, ordering, positioning, imaging, and so on. Windows can also handle standard user interactions—such as zooming the window to the appropriate monitor—automatically. For examples of the standard windows, see "Windows" (page 1-17).
- `HIPanel` is the other direct subclass of `HIObject`. It is the abstract superclass for all kinds of panels. `HIPanel` provides basic methods required by all panels for initialization, handling events, and getting information about the container hierarchy in which a particular panel is located. For an introduction to panels, see "Panels" (page 1-22).
- `HIAbstractList` is the abstract superclass for all lists. Its methods add, manipulate, and delete list items. Its concrete subclasses, `HIList` and `HIMenu`, define additional methods that perform operations specific to lists and menus. List panels and menu panels can also handle all user interaction, including input from a pointing device or keyboard, automatically. For examples of the standard panels you can create with these classes, see "Lists" (page 1-34) and "Menus" (page 1-32).
- `HIEditText` is the concrete class for editable text panels. Its methods perform text-specific operations such as inserting, deleting, and replacing text. Editable text panels can handle user interaction and text input automatically. For examples, see "Editable Text Panels" (page 1-35) and "Customizing HI Objects" (page 1-56).
- `HIControl` is the abstract superclass for all controls, including slider panels, scroll bars, pop-up buttons, progress indicators, and buttons such as push buttons and radio buttons. Its methods perform operations common to all controls, such as getting and setting control values. For examples of the standard controls, see "Controls" (page 1-24).

- `HIEmbeddingPanel` is the superclass for all panels that can contain embedded subpanels:
  - `HIRootPanel` implements the root panel, which is an embedding panel associated with every window. The window forwards events that affect the window's content to the root panel.
  - `HIDialog` is the superclass for all dialog boxes. In Mac OS 8, a dialog box is a specialized root panel that coordinates all user interaction with its subpanels. For an introduction to dialog boxes, see "Dialog Boxes and Alert Boxes" (page 1-28).
  - `HIModalDialog` is a subclass of `HIDialog` used to create modal dialog boxes.
  - `HIAlert` is a subclass of `HIModalDialog` used to create alert boxes.
- `HIScrollingPanel` implements a panel designed to contain any other panel (for example, a list panel or editable text panel) that is larger than the scrollable area allocated for the scrolling panel.
- `HIRadioButtonGroup` is the concrete class for radio button group panels that can automatically handle user interaction with the individual radio button panels in the group. For an example, see "Radio Button Groups" (page 1-36).
- `HIVisualSeparator` is the concrete class for panels that encapsulate horizontal, vertical, and rectangular visual separators. For examples, see "Visual Separators" (page 1-36).
- `HIStaticPanel` is the abstract superclass for panels that encapsulate icons of different bit depths, static text, QuickDraw pictures, and other images. The subclasses `HIIcon`, `HICaption`, and `HIPicture` provide a convenient way to integrate these purely visual elements with other HI objects. For examples, see "Static Image Panels" (page 1-37).

Some of these classes, such as `HIObject` and `HIWindow`, also define static methods, which are methods you can call without first specifiying a particular object. For example, you use static methods to instantiate and initialize an object using data in an AE record or to manipulate groups of windows.

The HI Objects class library provides three key benefits associated with OOP:

- **Inheritance** allows subclasses to share characteristics defined by classes above them in their branches of the class hierarchy. The HI Objects class library provides a wide range of standard subclasses that you can use to assemble complex human interfaces. If necessary, you can use OOP techniques to create your own subclasses without having to start from scratch.
- **Encapsulation** refers to the packaging of all the code that implements an HI object's appearance, state, and behavior (including theme-dependent characteristics) within the object itself, thus protecting it from inappropriate changes. Encapsulation also makes it possible to keep HI objects completely separate from the application behavior that they control. You don't have to subclass each time you want to use a particular HI object for a new purpose. Instead, you specify a callback function during instantiation that implements application-specific behavior when the object's state changes. This allows you to reuse the same HI object in completely unrelated applications just by specifying a new callback function.
- **Polymorphism** is the ability to call objects of different classes with the same method; for example, you can call the `Draw` method to draw an HI object of any class. Polymorphism permits the unification of disparate human interface elements in a single programming interface, which simplifies the construction of your application's human interface. You always trigger common behavior the same way no matter what kind of HI object is involved. Instead of learning how to implement similar behaviors in slightly different ways for different managers, you use the same method to implement the same behavior for a variety of objects.

HI objects also provide all the benefits of SOM objects, including the following:

- **Language independence.** You don't need to use an object-oriented language such as C++ to use HI objects. SOM supports a variety of object-oriented and procedural languages, and you can use the standard HI objects to create complex human interfaces with only minimal knowledge of OOP.
- **Binary compatibility.** Because SOM solves the "fragile base class problem," future versions of any HI object class can be released without breaking existing versions.

For more information about SOM and its benefits, see the accompanying document *SOM and Software Extensibility.*

Other benefits of using the HI Objects class library include the following:

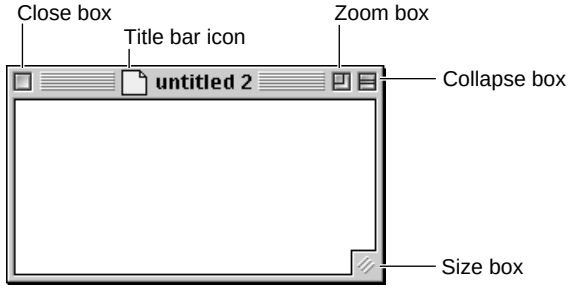
- **Embedding.** You can arrange standard HI objects in containment hierarchies by using various kinds of embedding panels and without subclassing any of the standard panels.
- **User input focus.** HI objects provide built-in support for user input focus, redrawing themselves as necessary when the focus changes and receiving and handling focused events appropriately. These capabilities automate much of the programming involved in controlling user input focus for HI objects in the same embedding panel.
- **Navigation events.** HI objects can automatically handle Navigation events when appropriate—that is, events that provide alternate access to HI objects with user input focus. For example, if a checkbox has user input focus, pressing the space bar repeatedly selects and deselects it; and if a slider has user input focus, pressing an arrow key moves the slider in the corresponding direction. Dialog box panels respond automatically to Navigation events (for example, an event generated by a Tab key press) that move user input focus from one subpanel to another.
- **Collection items.** You can attach collection items to any HI object. For more information about collections, see "Extensible Data Structures" (page 1-54).
- **Drag Manager support.** HI objects greatly simplify implementation of drag-and-drop behavior, such as dragging text from one editable text panel to another.

For a description of the superclass `HIObject`, see "HIObject Class Reference" (page 4-5). Information about Drag Manager support and references for the other classes in the HI Objects class library will be available with later developer releases.

Windows

Most applications use windows to present information to and interact with the user. Figure 1-4 shows a standard document window and its elements.

Figure 1-4 A standard document window



The window in Figure 1-4 includes the following elements:

- A **close box** that dismisses the window.
- A **title bar icon** that users can use as a proxy for the document's Finder icon in drag-and-drop operations. For example, the user can can drag a document's title bar icon to a particular folder on a volume, then drop it to save the docment in that location. For more information about title bar icons, see the accompanying document *Human Interface Guidelines for Mac OS 8.*
- A **collapse box** in the upper-right corner that users can click to control "windowshade" behavior—that is, to hide or show all of the window except the title bar.
- A **zoom box** next to the collapse box. Class `HIWindow` includes built-in support for monitor-specific zooming. Clicking the zoom box once causes the window to expand to its optimal size on the monitor on which most of its area is currently displayed. Clicking the zoom box a second time restores the window to its previous size and location.
- A **size box** in the lower-right corner that users can drag to resize the window.

You can use `HIWindow` methods to get and set various attributes of any window, such as whether it has a size box, collapse box, zoom box, title bar icon, or close box. `HIWindow` also supports resizing of windows in directions other than down

and to the right, the use of multilingual text in window titles, and other features that simplify localization.

Windows instantiated from class `HIWindow` can handle standard user interactions automatically, including resizing or moving the window and zooming it to the appropriate monitor.

For more information about multilingual text, see "International Text" (page 1-53). The chapter "Windows," which will be available with later developer releases, describes how to use the methods and static functions defined by `HIWindow`.

Window Layers

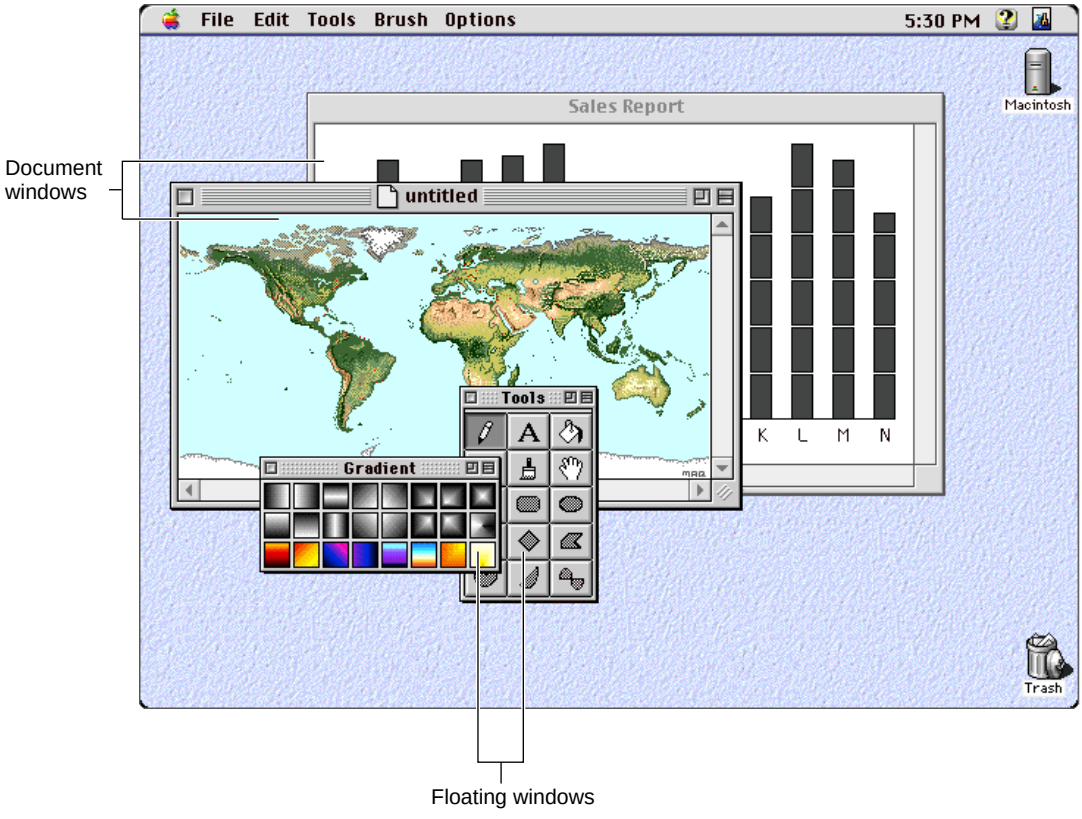
Every application has a **layer** within which it can display its windows. Various system services, such as Apple Guide and the Text Services Manager, control additional layers that may appear in front of or behind your application's layer.

Every window in an application's layer belongs to a "class" (not to be confused with OOP classes) that determines how it appears in relation to other windows in the same layer:

- **Modal windows** appear in front of all other kinds of windows in an application's layer. They are used for modal dialog boxes and alert boxes that require immediate attention from the user.
- **Floating windows** appear in front of document windows and behind modal windows in an application's layer. They are used for tool palettes, catalogs, and other elements used to act on data in document windows.
- **Document windows** (like that shown in Figure 1-4) appear behind floating windows and modal windows in an application's layer. They are used for document data such as graphics and text and (without the title bar icon or the size box) for modeless dialog boxes.

Windows of each class maintain this layering within a single application's layer. For example, the floating windows shown in Figure 1-5 always appear in front of the same application's document windows.

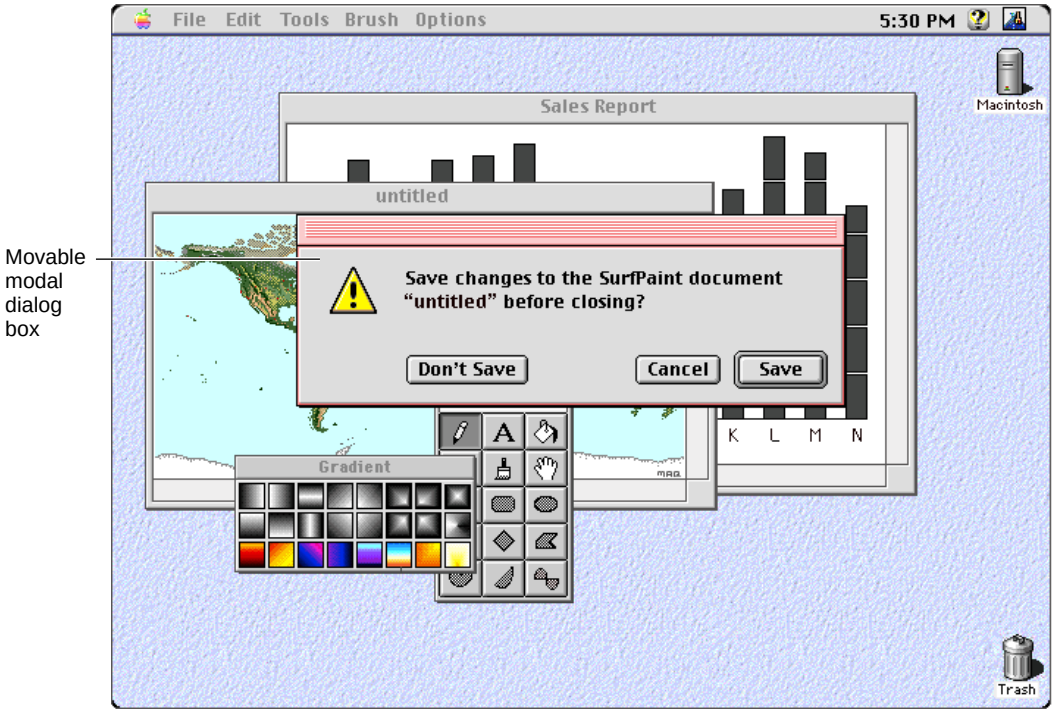
Figure 1-5 Layering of floating windows and document windows



A user typically has one or more windows open, often from several different applications. However, only one window can be the active window. An application's **active window** is the frontmost nonfloating window in the application's layer that is currently receiving user input. It is identified by distinctive details that aren't visible for inactive windows; for example, the Apple Default theme displays title bars for active windows with characteristic "racing stripes."

In Figure 1-5, the document window "untitled" is the active window. The other document window in the figure is inactive. If the user manipulates the floating windows, the corresponding actions affect contents of the active window.

Figure 1-6 Modal windows always appear in front of other windows in an application's layer.



When the user attempts to close the window "untitled" in Figure 1-5 without saving its contents, the application displays a movable modal dialog box in a modal window, as shown in Figure 1-6.

The modal window containing the dialog box appears in front of all other windows in the application's layer, so it is now the active window. Decisions made by the user with the aid of the dialog box apply to the document "untitled." The user can't manipulate the floating windows in this situation.

When your application displays any modal window, the menu bar automatically changes to a modal state, deactivating menus that aren't available when the modal window is displayed. The menu bar returns to its original state when the modal window is gone.

Windows instantiated from `HIObject` automatically support the guidelines for active and inactive windows as described in *Macintosh Human Interface Guidelines.* For example, they activate the scroll bars and highlight any selection when activated, and they deactivate the scroll bars and change or remove highlighting from any selection when deactivated. Similarly, an application's floating windows automatically disappear when the user switches to a different application.

If you are creating a window that needs to appear in a layer other than your application's primary layer, you can get a reference to the layer in which it should appear from the system service involved. For example, to create a window for a spelling checker managed by the Text Services Manager, you can get a reference to the text services layer from the Text Services Manager and then use that reference to initialize the window in that layer.

Window Groups

Window groups provide a useful abstraction for keeping your application's windows organized and automate many aspects of window management. You can add any of your application windows to a window group, regardless of window class. You can also add groups to groups.

You typically use a window group to associate a window with one or more additional windows, so that clicking the original window brings all of its associated windows to the front of their respective sublayers (while maintaining their ordering with respect to each other). For example, you might want to show the tool palettes associated with a document window whenever the user activates that window. To do so, you use `HIWindow` static functions that allow you to bundle the floating windows that contain the tool palettes with the document window as a unique window group. The tool palettes then come as far forward as they can (otherwise maintaining their current ordering) whenever the user activates the document window.

Your application itself has a group associated with it that contains all your application's windows. Thus, in addition to providing a means of associating related windows, window groups provide a mechanism for getting information about your application's windows and the order in which they are displayed in your application's layer.

For example, to find out which window is behind a given window, you first use an `HIWindow` static function to obtain a list of the windows in the application group in the order they are displayed in the application's layer. You can then iterate through the windows in the list to determine which window is behind

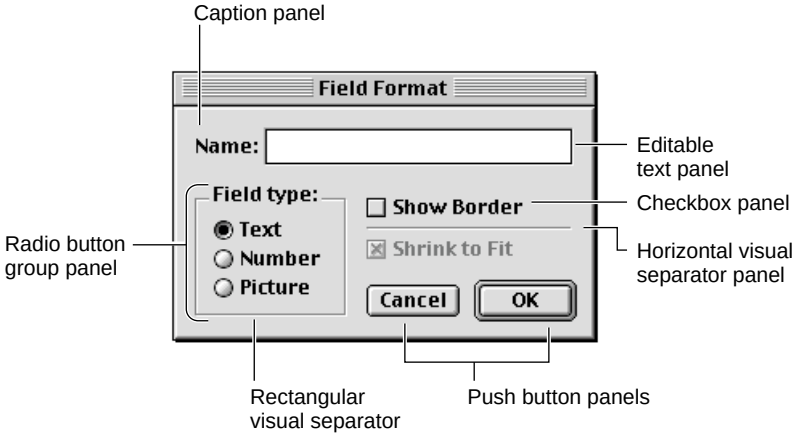
the one you're interested in. Similarly, you can get a list of windows in the application group to determine which windows need to be closed when the user quits the application.

Panels

Any HI object that can be placed in a window is a panel. A panel controls all aspects of its appearance and behavior as its state changes in response to user activities. You determine how changes in the panel's state affect your application.

Figure 1-7 illustrates a typical use of standard panels in a movable modal dialog box.

Figure 1-7 Standard panels used in a movable modal dialog box



All the panels in Figure 1-7 are embedded within the standard dialog box panel (class `HIDialog`), which is a root panel that encapsulates the entire contents of a dialog box. The dialog box panel isn't labeled in the figure; it consists of the entire content area of the modal dialog box. Like any other embedding panel, a dialog box panel controls the layout, user input focus, and user interaction for all the panels it contains.

These standard panels are embedded within the dialog box panel in Figure 1-7:

- **Caption panel.** This panel (class `HICaption`) provides an easy way to display static text.
- **Editable text panel.** This panel (class `HIEditText`) handles all user interaction within an editable text box.
- **Radio button group panel.** An embedding panel (class `HIRadioButtonGroup`) that encapsulates radio button controls. A radio button group panel controls user input focus and user interaction for its radio buttons, changing their highlighting and state as appropriate in response to user actions such as mouse clicks and key presses.
- **Visual separator panels.** Horizontal, vertical, and rectangular visual separator panels (class `HIVisualSeparator`) can be manipulated and positioned like any other panel within any embedding panel. You can also specify title text for the rectangular visual separator to display.
- **Checkbox panels.** An active checkbox panel (class `HICheckBox`) can select and deselect itself in response to user input such as mouse clicks; draw itself with and without user input focus; and select or deselect itself when it has user input focus and the user triggers an appropriate Navigation event, for example by pressing the space bar.
- **Push button panels.** The Cancel and OK push button panels (class `HIPushButton`) highlight themselves appropriately in response to user input such as mouse clicks, pressing Command-period, and pressing the Return or Enter key.

To associate a panel's state with a particular action or setting in your application, you specify a callback function for the panel to call when its state changes. For example, the dialog box panel Figure 1-7 specifies a state change function for the radio button group panel that enables the Shrink to Fit checkbox when the Picture radio button is selected and disables the checkbox when the Picture radio button is deselected. Similarly, the state change that occurs when a user chooses a menu command might invoke the function provided by your application that executes the command. This arrangement keeps the implementation of the panel itself separate from the specific behavior that your application associates with a particular object state.

The sections that follow include examples of the standard panels defined by subclasses of `HIPanel`.

Controls

Most windows and dialog boxes contain controls. Controls are HI objects that the user can manipulate with a pointing device, from a keyboard, or via other input devices to perform actions in your application or to change settings that modify future actions.

Table 1-1 shows examples of the panels you can create with the standard subclasses of `HIControl`. The appearance of each control is defined by its class and by the current theme.

Table 1-1 Some of the standard panels that encapsulate controls

Example in the Apple Default theme	Name and class	Description
	Push button panel `HIPushButton`	A button that displays an image (such as text, an icon, or a picture) indicating its purpose. Used to perform an instantaneous action when clicked by the user, such as completing operations defined by a dialog box or acknowledging an error message in an alert box. You can optionally specify that a push button is the default button, in which case it draws itself with the standard default appearance for the current theme (for example, with a ring around it) and responds when the user presses the Return or Enter key.
	Bevel button panel `HIBevelButton`	Like a push button, a bevel button displays an image (such as text, an icon, or a picture) indicating its purpose. Bevel buttons are commonly used in toolbars and palettes and to indicate state changes in dialog boxes. A bevel button can behave in one of two ways: animate momentarily (like a push button) or toggle back and forth between a pressed state and an unpressed state (similar to a checkbox).

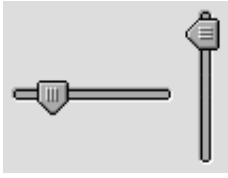
Table 1-1 Some of the standard panels that encapsulate controls (continued)

Example in the Apple Default theme	Name and class	Description
	Checkbox panel `HICheckbox`	A control that displays a small square with a label that may be text, an icon, a picture, or any other image indicating what kind of setting it controls. Used for an option that must be off, on, or in a mixed state. The square is checked when the setting associated with the box is on, is empty when the setting is off, or contains a short horizontal line when the setting is mixed. (For more information about the mixed state, see page 1-28.) Several adjacent checkboxes may be selected at the same time.
	Radio button panel `HIRadioButton`	A button that displays a circle with a label that may be text, an icon, or a picture beside it indicating what kind of setting the radio button controls. Like checkboxes, radio buttons retain and display an on-or-off setting; however, only one radio button in a group of radio buttons should be on at any one time. In the Apple Default theme, the circle is filled when the setting associated with the button is on, is empty when the setting is off, or contains a short horizontal line when the setting is mixed. (For more information about the mixed state, see page 1-28.)

Table 1-1 Some of the standard panels that encapsulate controls (continued)

Example in the Apple Default theme	Name and class	Description
	Disclosure triangle panel `HIDisclosureTriangle`	A triangle used to control progressive disclosure in lists, such as lists of files and folders in the Finder. When the arrow points right, only one item should be visible beside it. When the arrow points down, both the original item and the items contained within it should be visible in the list. To toggle between the two states, the user clicks the disclosure triangle, which turns with a characteristic animation defined by the current theme.
	Little arrows panel `HILittleArrows`	A pair of arrows that typically accompany a text box containing a numerical value, such as the date or time. Clicking the up arrow should increase the value in the text box, and clicking the down arrow should decrease it.
	Progress indicator panel `HIProgressIndicator`	A horizontal display used to indicate the progress of a lengthy operation (typically more than three seconds). If you don't know how long an operation will take, you can let the user know that it's still in progress by rotating an indeterminate progress indicator like a barber pole, as in the upper example. If you can supply values to the panel indicating how much of an operation has been completed, you can use a determinate progress indicator, which fills itself in from one end to the other to indicate what percentage of the operation has been completed, as in the lower example.

Table 1-1 Some of the standard panels that encapsulate controls (continued)

Example in the Apple Default theme	Name and class	Description
	Slider panel `HISlider`	Displays a range of values, magnitude, or position. A movable indicator shows the current setting. Sliders allow users to alter the value of the slider by moving the indicator up and down or back and forth. Sliders can be analog or digital devices that display their values graphically.
	Popup button panel `HIPopUpButton`	A button with an associated menu. When the user presses the mouse button while the pointer's over the popup button, additional menu items appear, as shown here. You can use a popup button as an alternative to a radio button group or a list.
	Scroll bar panel `HIScrollbar`	Windows and lists can have a horizontal scroll bar, a vertical scroll bar, or both. In the Apple Default Theme, a scroll bar is a narrow rectangle with an arrow in a box at each end and a scroll box that moves between them. Users can click the arrows or drag the scroll box to display more of the document by scrolling it into view. Scrolling should be live—that is, the contents of the window should scroll at the same time that the user is dragging the scroll box—as long as this doesn't degrade performance.

Radio buttons and checkboxes can be displayed in three different states: on, off, or mixed. A **mixed state** indicates that a setting is on for some elements in a selection and off for others. For example, a checkbox that determines whether a character is boldface appears in a mixed state if some characters in a range of selected text are bold and others aren't. The user can change a checkbox in a mixed state to either on or off for all the elements concerned, but can't directly change a checkbox that's on or off to a mixed state.

You can use `HIControl` methods to get a series of control values back from a control such as a slider or scroll bar while a user is still manipulating it. For example, you can get control values back from a scroll bar that allow your application to redraw the window's contents while the user is dragging the scroll box (live scrolling), or you can change the sound volume while the user is still manipulating the slider rather than waiting until the user releases it. All control values are 32-bit values, permitting manipulation of any control at a very detailed level of granularity.

The chapter "Controls," which will be available with later developer releases, describes how to create and manipulate the standard controls.

Dialog Boxes and Alert Boxes

Dialog boxes and alert boxes are specialized root panels that your application displays inside a window when it needs to interact with the user. In both cases, the panel controls all user interaction with its subpanels; for example, it tracks and updates user input focus and maintains its own modal state, if any. The dialog box or alert box panel also has a distinctive frame just inside the frame of the window in which it is displayed.

Dialog Boxes

An application displays a **dialog box** to solicit specific kinds of information from the user by means of the panels it contains, such as button panels, text panels, and list panels. In general, you should use either modeless or movable modal dialog boxes.

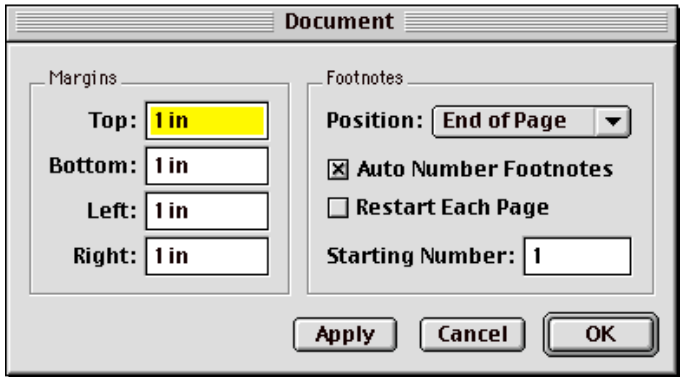
A **modeless dialog box** (class `HIDialog`) is a dialog box inside a document window that doesn't have a size box or scroll bars. Figure 1-8 shows an example.

Figure 1-8 A modeless dialog box

A modeless dialog box does not require the user to respond before doing anything else. The user can move a modeless dialog box, move between a modeless dialog box and other windows, and close a modeless dialog box just like a document window. Whenever possible, use a modeless dialog box instead of a movable modal dialog box.

A **movable modal dialog box** (class `HIModalDialog`) is a dialog box inside a modal window. It requires the user to work in a single mode within your application—that is, only inside the dialog box—until the completion of the user's interaction with that dialog box. Figure 1-9 shows an example.

Figure 1-9 A movable modal dialog box



A movable modal dialog box has a title bar (but no close box or size box) that allows the user to move it around the screen, for example to examine the part of the screen that it covers. The user can dismiss the dialog box only by clicking its buttons; however, the user should be able to switch layers by clicking in another application's window or by choosing another application from the Apple or Application menu. You should prevent the user from switching layers only if doing so risks immediate damage to the user's data.

It's also possible to instantiate a nonmovable modal dialog box from class `HIModalDialog`. A **nonmovable modal dialog box** resembles a movable modal dialog box except that it has no title bar and the user can't move it. `HIModalDialog` supports this form of dialog box for backward compatibility only. Mac OS 8 applications should use movable modal dialog boxes instead.

`HIModalDialog` takes care of the event handling required to enforce the modal state for both movable modal and nonmovable modal dialog boxes. For more information, see "Toolbox Support for Modal States" (page 2-25).

Alert Boxes

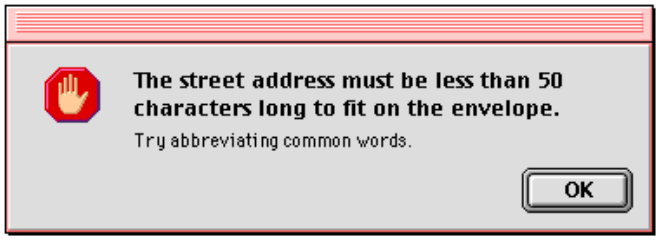
An application displays an **alert box** to warn or to report an error to the user. An alert box typically consists of an icon, text describing the situation, and buttons for the user to acknowledge or rectify the problem. You can use a note alert box, caution alert box, or stop alert box, each with a corresponding icon that indicates the seriousness of the information conveyed by its text. (For information about when to use which icon, see *Macintosh Human Interface Guidelines*.)

You use `HIAlert` to create alert boxes with the icon of your choice using a standard layout. In most cases, you should create movable alert boxes. It's also possible create a nonmovable alert box.

A **movable alert box** (class `HIAlert`), like a movable modal dialog box, has a title bar (but no close box) that allows the user to move it. However, a movable alert box can contain only text, an icon, and button panels, whereas a movable modal dialog box can contain any combination of panels. The user can dismiss a movable alert box only by clicking its buttons. As with a movable modal dialog box, you should normally allow the user to switch layers while a movable alert box is active by clicking in another application's window or by choosing another application from the Apple or Application menu.You should prevent the user from switching layers only if doing so risks immediate damage to the user's data.

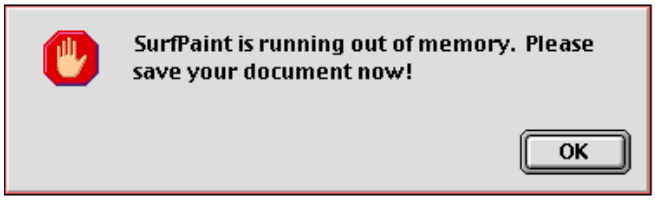
Figure 1-10 shows a movable alert box that the SurfWriter application displays if a user attempts to create an envelope with an address that's too long.

Figure 1-10 A movable alert box



A **nonmovable alert box** resembles a movable alert box except that it has no title bar and the user can't move it. Figure 1-11 shows an example. `HIAlert` supports this form of dialog box mainly for backward compatibility. In most cases you should use a movable rather than a nonmovable alert box. Use a modal alert box only when it's essential for the user to make an urgent decision immediately.

Figure 1-11 A nonmovable alert box



The chapter "Dialog Boxes and Alert Boxes," which will be available with later developer releases, describes how to create dialog boxes and alert boxes and provides human interface guidelines for using them.

Menus

A menu lets the user view or choose an item from a list of choices or commands that your application provides. You design your application's menus according to the tasks or actions your application performs. All applications support the Apple, File (or Document for OpenDoc part editors), and Edit menus and should add menu items as appropriate to the Help, Keyboard, and Application menus.

You can use class `HIMenu` to instantiate menu panels that

- contain polymorphic image data
- can be torn off
- display menu items in a grid
- display menu items in any font in any language using any script
- display keyboard equivalents for any menu item using multiple modifier keys
- support a "sticky menu" mode that allows users to leave a menu or submenu open and choose menu items by clicking them or from the keyboard
- respond automatically to user actions, such as highlighting menu items when the user navigates through them

You can also use `HIMenu` to show and hide the menu bar.

Figure 1-12 and Figure 1-13 show three menus created from `HIMenu`, including menu items that consist of pattern and color swatches, text items, dividers, and submenus.

To associate icons, text, pictures, patterns, and other simple visual elements with specific menu items, you create an image reference for that element with the aid of the appropriate imaging object, then pass the image reference to a method. For an introduction to imaging objects, see "Imaging Objects" (page 1-40).

The chapter "Menus," which will be available with later developer releases, describes in detail how to create menu panels.

Figure 1-12 Some standard menu items

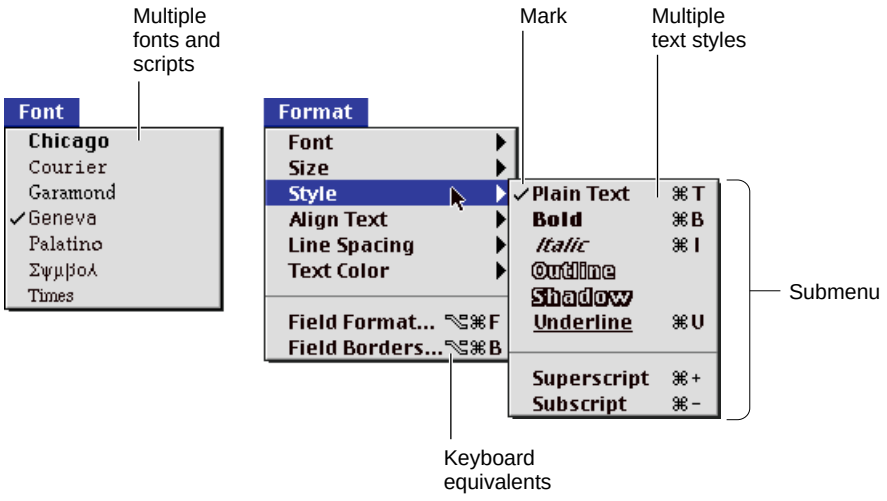
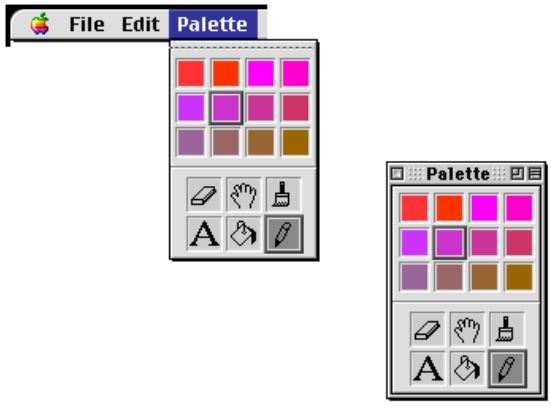


Figure 1-13 Standard tear-off menu with custom layout



Lists

A **list** is a series of items displayed within a rectangle. Each item in a list is contained within a rectangular **cell.** All cells within a list are the same size, but may contain different types of data and multiple columns of data. The user can click cells to select them.

You can use class `HIList` to instantiate list panels with cells that

- contain polymorphic image data
- display text images in any font in any language using any script
- respond automatically to user actions, such as highlighting when the user navigates through them

Figure 1-14 shows a list panel embedded in a scrolling panel.

Figure 1-14 A list panel embedded in a scrolling panel

To create a scrolling list like that shown in Figure 1-14, you instantiate a list from `HIList` and embed it in a scrolling panel instantiated from `HIScrollingPanel`. To arrange one or more lists with buttons and other controls in a window, you simply add the lists (or scrolling panels that contain lists) to an embedding panel.

You can use `HIList` methods to store and update the data within a list, display the list within a window with an appearance that matches the current theme, and respond appropriately to user interaction with a list. List panels store all offsets and values using 32-bit values, permitting the association of large amounts of data with a single list.

To associate icons, text, pictures, patterns, and other static images with specific list items, you first create an image reference for that element with the aid of the appropriate imaging object, then pass the image reference to a method. For an introduction to imaging objects, see “Imaging Objects” (page 1-40).

The chapter "Menus and Lists," which will be available with later developer releases, describes in detail how to create lists.

Scrolling Panels

A scrolling panel contains a vertical scroll bar, horizontal scroll bar, or both and is designed to contain any other panel (for example, a list panel or editable text panel) that is larger than the scrollable area allocated for the scrolling panel. Scrolling panels include methods that allow you to set and get the scrolled panel, vertical and horizontal scroll values, vertical and horizontal scroll increments, and scroll bar visibility.

You use `HIScrollingPanel` to instantiate a scrolling panel, to which you can then add the panel to be scrolled. For an example of a scrolling panel that contains a list, see Figure 1-14 (page 1-34).

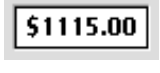
Editable Text Panels

A text panel displays the contents of a text object using the services of a **text engine,** which manages the formatting, drawing, and editing of the text in response to user actions and calls to panel methods. When you instantiate a text panel, you specify the text engine you want it to use. Mac OS 8 provides a default text engine based on TextEdit in System 7, and you can provide or use other text engines according to the needs of your application.

The text panel itself is independent of the text engine with which it is associated. Although its methods can perform text-specific operations such as inserting, deleting, and replacing text, the text panel can also use its associated text engine to respond automatically to user input from the mouse, keyboard, voice-recognition software, and other sources of input. A text panel can also support copy, paste, and drag and drop automatically.

You instantiate editable text panels like that shown in Figure 1-15 from class `HIEditText`. An editable text panel permits editing by the user of the text that the panel displays.

Figure 1-15 An editable text panel



It's possible to disallow editing temporarily in an editable text panel. This can be useful, for example, for panels that are editable only some of the time, depending on other conditions in the application. For an example of an editable text panel that can be temporarily disabled, see Figure 1-23 (page 1-55).

The easiest way to create static text panels whose text never needs to be editable is to use `HICaption`. For an example of a static text panel, see Figure 1-20 (page 1-39).

For information about text objects, see "International Text" (page 1-53). For information about text engines, see the accompanying document *Text Handling and Internationalization.*

Radio Button Groups

A radio button group is an embedding panel that encapsulates several radio button panels, as shown in Figure 1-16. Unlike the individual radio button panel illustrated on page 1-25, a radio button group panel can handle mouse and keyboard interaction, including highlighting and tracking user input focus. You use `HIRadioButtonGroup` to instantiate a radio button group.

Figure 1-16 A radio button group panel

Visual Separators

Visual separator panels can display horizontal, vertical, or rectangular visual separators. Figure 1-17 shows examples of horizontal and visual separators. A rectangular visual separator can optionally include a title. For an example of a rectangular visual separator panel, see Figure 1-7 (page 1-22).

Figure 1-17 Horizontal and vertical visual separator panels

You use `HIVisualSeparator` to instantiate a visual separator panel. To display visual separators that aren't part of a panel, you can use the Appearance Manager primitives shown in Table 1-2 (page 1-50).

Static Image Panels

Embedding panels such as dialog boxes and palettes often include icons, pictures, patterns, and simple unstyled captions. Although users don't interact with these elements, it is often convenient to implement them as panels.

The standard simple visual panel classes provide the easiest way to integrate visual elements with your application's interactive human interface elements. You set the title for a simple visual panel—that is, the visual element to be displayed—by passing data of the appropriate type to its initializing method, which automatically creates the appropriate imaging reference and displays it using imaging object methods.

You can also implement simple visual elements without the aid of the HI Objects class library by using imaging objects or other lower-level services directly. For an introduction to imaging objects, see "Imaging Objects" (page 1-40).

The chapter "Static Image Panels," which will be available with later developer releases, describes in detail how to use `HIStaticPanel` subclasses to create and manipulate visual images as panels.

Icons

An icon is a graphic representation of some human interface element, such as a document, disk, folder, application, or the Trash in the Finder. The Finder draws and manages the icons that a user sees on the desktop.

Figure 1-18 shows the talking face icon commonly used to identify note alert boxes.

Figure 1-18 An icon panel

To display an icon as a panel, use `HIIcon`. An icon panel encapsulates an icon image and can draw itself appropriately within an embedding panel.

To display an icon without using the HI Objects class library, use either the icon imaging object interface or the Icon Utilities, a lower-level set of utilities for manipulating icons that aren't inside panels. For example, you can use the icon imaging object interface to display icon images in any content area. The Icon Utilities provide low-level support for icon caching and drawing. You can also use the Icon Utilities to obtain the icon currently being used by a particular file in the Finder so you can display it in your application.

Pictures

To display a QuickDraw picture as a panel, use `HIPicture`. Figure 1-19 shows an example.

Figure 1-19 A picture panel

To display a picture outside of a panel, you can use either the picture imaging object interface or QuickDraw directly.

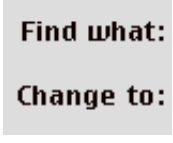
Captions

To display the contents of a text object as a static text panel that can't ever be edited by the user, use `HICaption`. Figure 1-20 shows two static text panels.

To create text panels that can be edited (or have the potential to be edited), use `HIEditText`. For an example of an editable text panel, see "Editable Text Panels" (page 1-35).

`HICaption` provides the easiest way to display static text in a panel. Unlike `HIEditText`, `HICaption` doesn't require that you specify a text engine.

Figure 1-20 A caption panel



To display the contents of a text object outside of a panel, use either the text imaging object interface or a text engine directly.

Both `HICaption` and text imaging object methods use **text strikes,** which are standard drawing contexts defined by QuickDraw for specifying the font, size, fractional widths, and other characteristics with which to draw text. Text strikes also allow you to specify abstractions for the current application font, large system font, small system font, and other special fonts that users can set themselves. In most cases you should specify either the large system font or the small system font for text that appears in your application's HI objects.

More information about text strikes will be provided with later developer releases.

Imaging Objects

HI objects support polymorphic image data for their titles and other elements by using imaging objects. An **imaging object** is a SOM object that can draw a specific kind of image data, such as text, icons, or pictures. The Imaging Objects class library defines a separate imaging object class for each of several commonly used types of image data. These classes descend from a common abstract superclass, `HIImagingObject`, as shown in Figure 1-21.

Unlike imaging objects, which are full-fledged SOM objects, the image data on which the imaging objects operate are not real objects in the OOP sense, since they don't have any methods of their own. The image data for a single image, such as a picture or some static text, is identified by an **image reference** that you can pass to imaging object methods or to HI object methods.

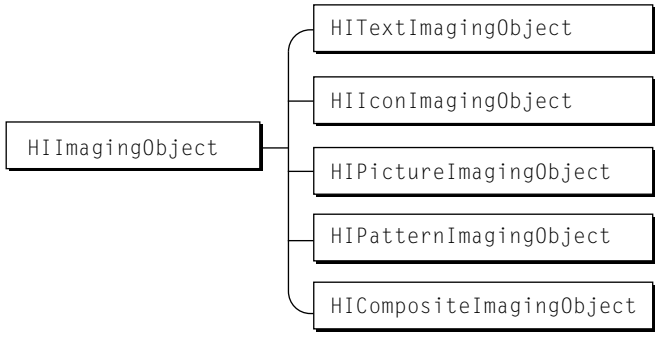
You can also bundle two or more images of potentially different types together as a single composite image, which is identified by a single image reference and drawn by a **composite imaging object** (class `HICompositeImagingObject`). For example, the combination of a file or folder icon accompanied by a text name, as displayed by the Finder, could be drawn by a composite imaging object.

You use image references with the HI Objects class library to specify

- titles of windows, push buttons, checkboxes, icons, rectangular visual separators, and all other HI objects that can have a title
- list items
- menu items

You can also pass an image reference to HI imaging object methods to measure or draw an image without using the HI Objects class library at all.

Figure 1-21 The inheritance hierarchy for the HI Imaging Objects class library



The classes shown in Figure 1-21 play the following roles:

- `HIImagingObject` is the abstract superclass for all HI imaging object classes. Its methods perform operations common to all imaging objects, such as creating, initializing, measuring, and drawing a new image.
- `HITextImagingObject` is the concrete class that defines text images. Its methods allow you to set and get the text object associated with an image and the text strike that defines the font, style, and other characteristics of the text to be displayed.
- `HIIconImagingObject` is the concrete class that defines icon images. Its methods allow you to set and get the icon image.
- `HIPictureImagingObject` is the concrete class that defines picture images. Its methods allow you to set and get the QuickDraw picture associated with a picture image.
- `HIPatternImagingObject` is the concrete class that defines pattern images. Its methods allow you to set and get the pattern.
- `HICompositeImagingObject` is the concrete class that defines composite images. Its methods allow you to add subimages to a composite image and to set and get image references for the subimages.

Copy, Paste, Drag, and Drop

Users of Mac OS 8 applications should be able to copy and paste data freely within the same window or from one window to another. Users should also be able to drag and drop anything that they can copy and paste. For example, the should be able to press down on the mouse button while the pointer is over a selection, move the pointer across the screen, and then release the mouse button.

You can use the Scrap Manager, Clipboard Manager, and Drag Manager to implement copy, paste, and drag and drop with the same piece of code. The Scrap Manager provides the generic transport package format, and the Clipboard and Drag Managers support the underlying user experience of moving the packaged data, called a *scrap,* from one place to another.

Because they don't involve drawing to the screen, the Scrap Manager and the Clipboard Manager execute in a preemptively safe manner and are fully reentrant—unlike the Drag Manager and the rest of the Mac OS 8 Toolbox, which are nonreentrant.

The chapter "Scrap, Clipboard, and Drag Managers," which will be available with later developer releases, describes in detail how to implement copy, paste, and drag and drop. The sections that follow introduce these three managers.

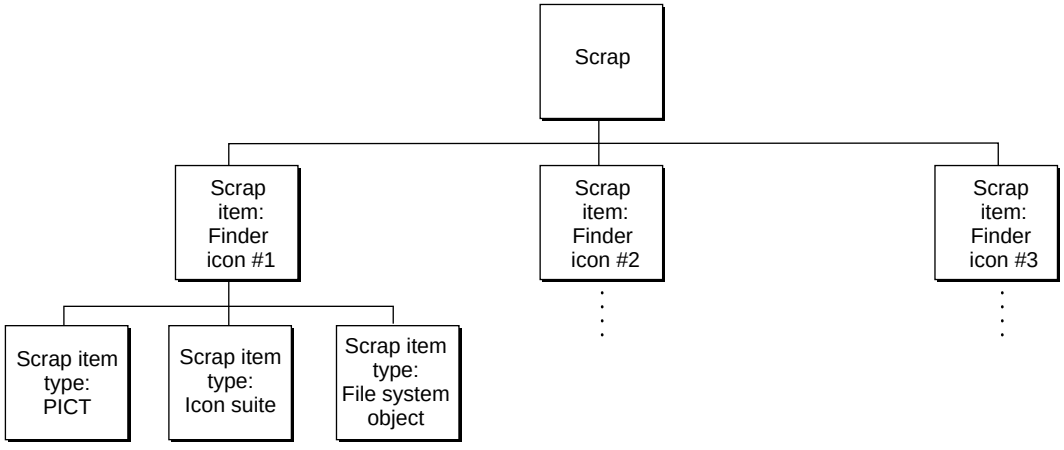
Scrap Manager

The Scrap Manager can handle data of any size, including QuickTime movies, sound data, graphics data, and other data that take up a lot of memory. The Scrap Manager provides functions that allow you to create a scrap, package the data to be transported inside it, and retrieve the data after the Clipboard Manager or Drag Manager have transported the scrap to its destination.

A **scrap** consists of one or more **scrap items.** Each scrap item is associated with a single piece of data represented by one or more **scrap item types.** For example, a scrap that contains a picture might contain a single scrap item represented by a single scrap item type, such as a PICT; whereas a scrap that contains a Finder icon might contain a scrap item represented by several scrap item types, including the icon itself, a PICT, and a file system object, as shown in Figure 1-22. It is usually desirable to provide the same data in several

different formats, so that the receiving application can choose a format that it can handle.

Figure 1-22 Scrap, scrap items, and scrap item types



The Scrap Manager also supports the concept of **promises.** For example, you can choose to put a scrap item on the Clipboard as a promise instead of the actual data. This involves constructing an empty scrap with placeholders for the various scrap item types. When the user pastes, the Scrap Manager sends a Scrap Promise event to the original application, requesting that it fulfill its promise for the type of data being pasted by providing the actual data. This mechanism avoids data transfer until the data is actually needed for a paste. It also allows the original application to transfer the data using just the format requested by the pasting application rather than duplicating the data in a variety of possible formats. Promises are especially useful for copying large pieces of data, but they are also the fastest way to copy any kind of data.

Promises placed on the Clipboard require slightly different treatment than promises used in dragging operations. If the user quits the original application or closes the document containing the promised data before a promise on the Clipboard has been fulfilled, the original application should fulfill the promise before it quits or closes the window to ensure that user can paste the item in the future.

Clipboard Manager

The Clipboard Manager provides a mechanism for placing a scrap on the Clipboard and retrieving it when the user pastes the data in a new location. A single Clipboard is shared by all currently running applications.

In general, after you use the Clipboard Manager to place a scrap on the Clipboard, the scrap becomes read only. Multiple applications can use the Scrap Manager simultaneously to extract data from the scrap during separate paste operations, but the scrap can't be altered. If another scrap gets placed on the Clipboard before one or more applications have finished pasting, the Scrap Manager maintains the old scrap until they are finished, but treats the most recent scrap as the current scrap for any new paste operations.

Whenever a new scrap gets placed on the Clipboard, the Clipboard Manager sends a Clipboard Changed Apple event to all interested applications to notify them that the contents of the Clipboard have changed and what data types are available for the new data. This allows each application to update its Edit menu appropriately as soon as new data is copied to the Clipboard. For example, when your application receives an event informing it that the Clipboard now contains text in a format your application can handle, it should make sure that the Paste item in its Edit menu is enabled.

Drag Manager

From the user's point of view, to **drag** something means to position the pointer on a visual interface element (such as an icon in the Finder), press and hold the mouse button, move the pointer to a new position, and then release the mouse button. In general, dragging can have different effects, depending on what's under the pointer when the user first presses the mouse button. These can include selecting blocks of text, choosing a menu item, selecting a range of objects, shrinking or expanding an object, or moving an icon or other visual elements from one place to another. The Drag Manager supports the latter form of dragging: moving visual elements and their associated data from one place to another.

You use the Drag Manager to support the dragging of visual interface elements within your application, from your application to other applications or the Finder, and from other applications or the Finder to your application. The Drag Manager uses a scrap to hold the data associated with a dragged element. Elements that may be dragged can include text, graphics, bitmaps, icons, outline items, and so on. The Finder itself uses the Drag Manager to support

common dragging operations such as moving a file or folder or dropping items on other items to make something happen, such as running a script.

The Scrap Manager packs and unpacks the data that's transported in a drag operation. The Drag Manager supports the user experience while an item is being dragged, including displaying an outline or a transparent version of the original image during dragging. You use the Drag Manager to create the scrap associated with a dragged element, the Scrap Manager to add scrap items and their associated scrap item types to the scrap, the Drag Manager to handle the actual dragging, and the Scrap Manager to read the scrap after the drag operation is complete.

The Toolbox provides default handlers for drag tracking and drag handling events that the Drag Manager uses to inform your application as the user drags items across the screen or drops drag items in one of your application's windows. If necessary, you can override these handlers to implement custom dragging behavior for your application.

Interactions With the Finder

Once you've designed your application, you need to create icons to represent the application and the documents it creates. The Finder displays these icons to the user. If your application appears as an item in the Apple or Application menu, your application's icon is displayed next to its name and, when your application is active, as the title of the Application menu.

Many applications allow users to set various preferences, such as the default font, pen widths, menu contents, toolbar contents, backup saving behavior, and so on. The Preferences Manager provides a standard mechanism for controlling your application's preferences. Using the Preferences Manager ensures that your application can take advantage of Mac OS 8 support for multiple users who share a single computer.

The chapter "Finder Interface" describes how to define and create the icons for your application and its documents. The chapter also describes how your application interacts with the Finder. The chapter "Preferences Manager" describes how to implement user preferences for your application. Both chapters will be available with later developer releases.

Resources

Resources are basic elements of every Macintosh application. By storing descriptions of menus, windows, controls, dialog boxes, sounds, fonts, and icons in resources, you can make these and other elements easier to create and manage. Using resources also eases translation of human interface elements into other languages.

A **resource** is any data stored according to a defined structure in the resource fork of a file. The data in a resource is interpreted according to its resource type. You usually create resources using a resource compiler or resource editor. This book shows resources in Rez format; Rez is a resource compiler provided with the Macintosh Programmer's Workshop (MPW), available from the *Apple Developer Catalog*. Apple and third parties also provide additional resource tools you can use to create the resources for your application.

Most of the Toolbox services use the Resource Manager to read resources for you. For example, you can use the `HIObject` static functions to read descriptions of your application's windows, dialog boxes, menus, and controls from resources. The Toolbox services interpret a resource's data for you once it is read into memory. To ensure compatibility with future versions of the Mac OS, you should use Toolbox services to access resources whenever possible. If necessary, you can also use the Resource Manager directly to read and write resources whose formats are defined by your application.

The chapter "Resource Manager," which will be available with later developer releases, describes the Resource Manager in detail. This section provides a brief introduction to resources in Mac OS 8.

Mac OS 8 treats a **file** as a named, ordered sequence of bytes that is stored on a volume and is typically divided into two forks, the data fork and the resource fork. The **data fork** contains data that usually corresponds to data created by the user; the application creating the file can store and interpret the data in the data fork in whatever manner is appropriate. The **resource fork** of a file consists of the resources themselves.

When you write data to a file, you write to either the file's resource fork or its data fork. You must use File Manager routines to read from and write to a file's data fork and Resource Manager routines to read from and write to a file's resource fork.

You typically store as resources data that has a defined structure—such as icons, sounds, or AE records that describe HI objects. When you create a resource, you assign it a resource type and resource ID. A **resource type** is a sequence of four characters that uniquely identifies a specific type of resource, and a **resource ID** identifies by number a specific resource of that type. (You can also use a resource name in place of a resource ID to identify a particular resource within a resource type.) For example, to create a description of a picture in a resource, you create a resource of type `'PICT'` and give it a resource ID or resource name that is unique among any other `'PICT'` resources that you have defined. Some resources have restrictions on the numbers you can use for resource IDs; in general, numbers 128 through 32767 are available for your use.

Mac OS 8 defines a number of standard resource types. You can use these resource types to define their corresponding elements. You can also create your own resource types if your application needs resources other than the standard types.

When your application or a Toolbox service requests a resource of a particular type with a given resource ID, the Resource Manager looks for the specified resource and, if successful, reads it into memory. However, the Resource Manager does not interpret the format of an individual resource type. You should not make any assumptions about a standard resource's format once the Resource Manager has read it into memory. For example, when you use a static function defined by `HIObject` to read a window from an HI object resource, the function uses the Resource Manager to read the resource into memory. Once the resource is in memory, the HI Objects class library interprets the resource's data and creates a window with the characteristics described by the resource. In general, you should not directly access resources in memory. The only exceptions are resources whose formats you define yourself.

You typically store the resources specific to your application—such as descriptions of its HI objects—in the application file's resource fork. Whether you store data in the data fork or the resource fork of a document file depends largely on whether you can structure that data in a useful manner as a resource. Data that is likely to be edited by the user is usually stored in the data fork of a document file. Document-specific settings, such as the document window's last position and size on the screen, are usually stored as a resource in the document file's resource fork. The next time the user opens the document, your application can read the position and size saved in this resource and position the document accordingly.

You can specify that the Resource Manager read a resource into memory immediately when the Resource Manager opens a file's resource fork, or you

can specify that the Resource Manager read it into memory only when needed. Normally, the Resource Manager stores resources from resource forks opened by your application in relocatable blocks in your application's heap. You can also specify whether the resource should be purged from memory to make room in memory for other data. If you specify that a resource is purgeable, you need to use the Resource Manager to make sure the resource is in memory before accessing it.

When a user opens your application, your application's resource fork is opened automatically. When your application opens a file, your application typically opens both the file's data fork and the file's resource fork. When your application requests a resource from the Resource Manager, the Resource Manager follows a specific search order. (If necessary, your application can change the search order using Resource Manager routines.) The Resource Manager normally looks first for the resource in the resource fork of the last file that your application opened. So, if your application has a single file open, the Resource Manager looks first in that file's resource fork. If the Resource Manager doesn't find the resource there, it continues to search each resource fork open to your application in the reverse order that the files were opened. After looking in the resource forks of files your application has opened, the Resource Manager searches your application's resource fork. If it doesn't find the resource there, it searches system resources.

This search allows your application to use system resources, to override system resources with resources stored in the application's resource fork, and to override application-defined resources with resources stored in a document's resource fork.

A resource fork can contain at most 2700 resources. In general, you should not create more than 500 resources of the same type in any one resource fork.

Themes

As shown in Figure 1-1 (page 1-4) and Figure 1-2 (page 1-5), users can select different themes, or styles—that is, coordinated sets of human interface designs that determine the appearance of human interface elements on a systemwide basis, across multiple applications. Regardless of the theme, the core user experience remains the same, and users can switch themes without having to learn new human interface metaphors. The Appearance Manager provides the underlying support for these capabilities.

A theme determines the appearance of all HI objects on the screen, including alert icons, controls, background colors, dialog boxes, menus, windows, and state transitions. Apple supplies several standard themes. Users can choose among the themes available to the system with the Appearance control panel, which also allows them to modify other aspects of their computing environment's appearance, such as the desktop pattern, highlight color, screen saver, and system font. (The Appearance control panel replaces the Desktop Patterns and Color control panels used in System 7.)

In addition to supporting user customization, themes insulate your application from future changes to the human interface. They free you from relying on hardwired appearances for standard elements while making it easier to create customized elements. Because Mac OS 8 allows you to deal with appearance abstractions rather than specific details, your application can support not only the new human interface designs in Mac OS 8 but also future design enhancements.

The Appearance Manager manages all aspects of themes and theme switching, the Appearance control panel, support for a variety of color data (RGB colors, pixel patterns, and so on), and support for animation and sound. It supersedes System 7 color tables such as `'cctb'` and `'mctb'` with a more abstract mechanism that allows you to coordinate colors with the current theme.

The Appearance Manager provides primitives for specifying window headers, group boxes, separators, and other building blocks for HI objects. Table 1-2 shows preliminary designs for some of the primitives provided by the Appearance Manager as they appear in the Apple Default theme. The HI Objects class library uses these primitives to draw HI objects. If you need to customize any of the standard HI objects, you should use these primitives to coordinate the appearance of your objects with the current theme.

Table 1-2 Some Appearance Manager primitives and examples of their use

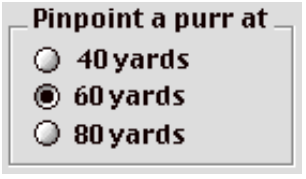
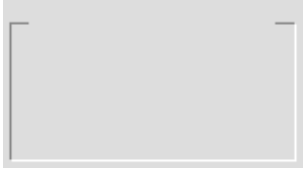
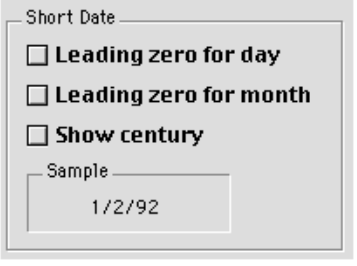
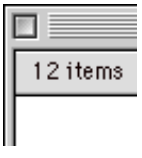
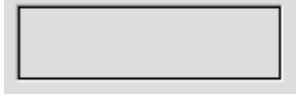
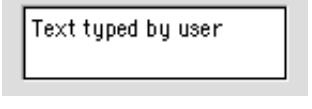
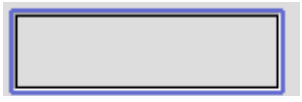
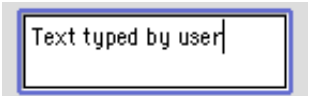
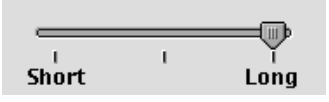
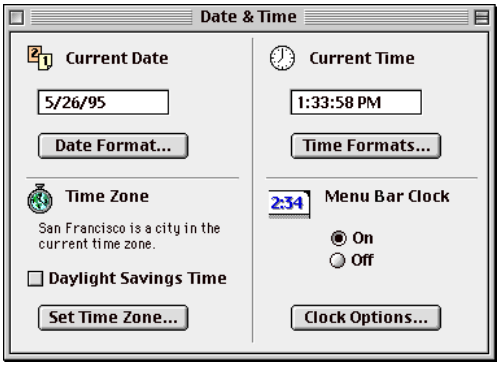
Appearance in the Apple Default theme	Example of use	Description
		Primary group box. Used to frame a primary group of related controls; title optional.
		Secondary group box. Used to frame a group of related controls within a primary group box; title optional.
		Placard. Used as background for status information in a window.
		Window header. Used to display information at the top of the content area, below the title bar.
		Text box frame. Used to enclose a text box.

Table 1-2 Some Appearance Manager primitives and examples of their use (continued)

Appearance in the Apple Default theme	Example of use	Description
		Focused text box frame. Used to indicate that a text box has user input focus.
		List box frame. Used to enclose a list. (Focused appearance is the same as a focused text box frame.)
		Ticks. Used to calibrate a slider or progress indicator.
 		Vertical and horizontal visual separators. Used to separate elements in a dialog box or window.

In addition to the primitives illustrated in Table 1-2, the Appearance Manager provides functions that allow you to determine how the current theme is drawing various aspects of the human interface, such as background fills. For example, you can ask the Appearance Manager for the current background color so you can coordinate the appearance of your application's content area with the current theme. Similarly, if you want to draw a line through a standard menu item, you can ask the Appearance Manager for the current color of the menu item text so you can use the same color for the line.

For more information about using the Appearance Manager to extend the HI Objects class library, see "Customizing HI Objects" (page 1-56). The chapter "Appearance Manager," which will be available with later developer releases, describes the Appearance Manager in detail.

Programming With the Toolbox

All Toolbox services support similar capabilities in similar ways, thus ensuring a consistent programming interface as well as a consistent user experience. From a programmer's point of view, these are the most important principles that underlie the Mac OS 8 Toolbox:

- **Opacity and consistency.** The Toolbox provides a complete programming model that doesn't require direct manipulation of underlying data structures.
- **Integrated support for international text.** The Toolbox takes advantage of Mac OS 8 text objects to provide flexible support for multilingual text throughout the human interface.
- **Object lifecycle management.** The Toolbox keeps track of multiple references to a single HI object on your application's behalf, releasing the original object only after all references to it have been released.
- **Data extensibility.** The Toolbox uses the Collection Manager to support the addition of arbitrary tagged data to Toolbox data structures without manipulating the structures directly.
- **Design extensibility.** The windows, menus, controls, and other standard elements defined by the HI Objects class library can be used as is or extended by developers to support specialized application needs.

The sections that follow introduce these four aspects of Toolbox programming.

Opacity and Consistency

The Mac OS 8 Toolbox provides high-level interfaces that eliminate the need to keep track of the internal organization of system data structures. Instead, Toolbox services ensure the opacity and consistency of the programming interface. The Toolbox provides

- the HI Objects class library for creating standard, customizable HI objects
- the HI Imaging Objects class library for creating standard image references that can be used to draw a variety of image types
- accessor functions or methods for getting and setting the contents of individual data structures
- blind references that identify data structures without permitting direct access
- high-level interfaces for all operations, including those traditionally associated with low-memory globals

The opacity the Toolbox interfaces ensures that your application doesn't have to depend on Toolbox implementation details. This allows Apple to develop the Toolbox further in the future without requiring you to rewrite your application.

International Text

Mac OS 8 supports a systemwide text data type, called a **text object,** that encapsulates the details of text encoding. Text objects allow applications to manipulate multilingual text transparently without dealing with the details of character encoding, which can be based on Unicode, ASCII, traditional Macintosh, and other encoding systems. Mac OS 8 applications should use text objects rather than Pascal and C strings within all human interface elements, including objects instantiated from `HIEditText` or `HICaption` and image references created with HI imaging objects.

Pervasive support for text objects in Mac OS 8 has two ramifications for application programming:

- You can display multilingual text (in multiple scripts) as the title of any HI object or as a menu item or list item.
- Because you don't have to keep track of the details of individual scripts and encoding systems, localization of interface elements is greatly simplified.

In addition to text objects, the Mac OS 8 Toolbox supports left-growing windows, automatically resizable dialog boxes, and other features that address specific international needs.

For more information about international text and Mac OS 8, see the accompanying document *Text Handling and Internationalization*.

Object Life Cycle Management

The Mac OS 8 Toolbox allows you to obtain multiple independent references to a single HI object. It does so by keeping track of all references to an HI object, incrementing the reference count when a new reference is created and decrementing it when a reference is released.

You always release HI object references the same way, regardless of how many references a single HI object might have, thus avoiding the programming involved in keeping track of them yourself. The Toolbox releases the original object from memory only when the reference count reaches zero.

Extensible Data Structures

The Mac OS 8 Toolbox eliminates the need for hardwired modification of system data structures by supporting the Collection Manager, which allows you to attach arbitrary data to virtually any data structure. The Collection Manager, which originally shipped with QuickDraw GX, can be used to associate data with a tag and ID, attach that data as a collection item to any Toolbox data structure, and retrieve it when necessary.

Collection items can be used for a variety of purposes. For example, in a preferences dialog box that allows the user to switch among several preference "pages," each of which displays multiple panels, you can use collection items to associate a page ID with the panels that appear in that page. This makes it easy to hide or show the appropriate panels when the user switches pages.

The original Collection Manager is described in *Inside Macintosh: QuickDraw GX: Environment and Utilities.* Information about the Mac OS 8 Collection Manager will be available with later developer releases.

Extensible Designs

Whenever possible, you should use the standard HI objects defined by the HI Objects class library. This is the easiest way to support themes. If you need to create custom HI objects, you have three choices:

- **Add standard HI objects to a standard embedding panel.** You can easily add standard HI objects to a standard embedding panel without subclassing.
- **Subclass your own theme-compatible HI objects.** You can subclass from any HI object class to create your own custom HI objects, if necessary using primitives and fills defined by the Appearance Manager to define new appearances that maintain support for theme switching.
- **Subclass your own HI imaging objects.** You can subclass from `HIImagingObject` to support additional image types that you want to use for HI object titles, menu items, list items, and so on.

The sections that follow briefly describe these three ways of extending the standard HI object designs. These techniques will be described in more detail in later developer releases.

Assembling Embedding Panels

The easiest way to create custom HI objects is to add standard HI objects to a standard embedding panel. Figure 1-23 shows an example.

Figure 1-23 Custom panel created by embedding a checkbox panel and editable text panel in an embedding panel

Disabled

Enabled with user input focus

The panel shown in Figure 1-23 is an embedding panel that contains two subpanels: an editable text panel and a checkbox that allows the user to enable or disable the text panel.

To create the panel in Figure 1-23, you can follow these steps:

1. Instantiate an embedding panel, checkbox panel, and editable text panel from the corresponding HI object classes.
2. Add the checkbox panel and editable text panel to the embedding panel as subpanels in the appropriate locations.
3. Set a state change function for the checkbox panel that enables the editable text panel when the user selects the checkbox. When the user deselects the checkbox, the state change function disables the editable text panel.

You can also specify such an arrangement in a description of the embedding panel that you can store in a resource.

The checkbox panel changes its appearance appropriately in response to user input such as mouse events or pressing the space bar. The editable text panel draws itself appropriately when it receives user input focus and supports standard user operations on text.

Customizing HI Objects

Figure 1-3 (page 1-12) shows the inheritance hierarchy for the standard classes defined by the HI Objects class library. You can use standard SOM techniques to subclass custom HI objects from any of these standard classes.

For example, you can create a custom editable text panel that accepts numbers but not letters. To do so, you subclass from `HIEditText` and override just two methods: `ReplaceTextByTextObject` and `HandleAppleEvent`.

If you need to change the appearance of a standard HI object, you can subclass from it and use Appearance Manager primitives to create the appearance you want in a theme-compatible manner. You can use this approach for a wide range of customization, from minor adjustments to the appearance of a push button to a completely new HI object subclassed from the `HIObject` superclass that you build entirely with Appearance Manager primitives.

Customizing HI Imaging Objects

Figure 1-21 (page 1-41) shows the inheritance hierarchy for the standard classes defined by the HI Imaging Objects class library. You can use standard SOM techniques to subclass custom HI imaging objects from the abstract superclass `HIImagingObject`.

For example, if your application uses a proprietary graphics format, you can subclass from `HIImagingObject` to implement an imaging object that draws images in that format. You can then pass image references for images in your proprietary format to any HI object methods that accept image references.

Toolbox Event Routing

Contents

The Toolbox defines a variety of standard events for which it provides default handlers. The Toolbox also provides default handlers for other events, such as key events and text events, generated by other parts of the system.

This chapter describes how some of the default handlers that the Toolbox provides route events to the appropriate HI objects. It also describes the points at which you can override the default event routing to implement your application's unique behavior.

Before you read this chapter, you should be familiar with the accompanying document *Apple Events in Mac OS 8* and with the first chapter in this document, "Introduction to the Mac OS 8 Toolbox" (page 1-3).

For an introduction to the treatment of text in Mac OS 8, see the accompanying document *Text Handling and Internationalization.*

For descriptions of some of the events for which the Toolbox provides handlers, see "Standard Events Handled by the Toolbox" (page 3-5).

▲ **WARNING**
This document is preliminary and incomplete. It is intended only to illustrate the design concepts that underlie Toolbox event routing. All information presented here, including the roles of individual events, handlers, and methods, is subject to change. ▲

Event Routing Within a Process

To dispatch Apple events within a process, the Apple Event Manager uses one or more Apple event dispatchers, which combine an event queue and a stack of handler tables. Dispatchers at the process level are called **process dispatchers.** Every process has a default dispatcher, and you may create additional process dispatchers as necessary. Every window has a single dispatcher called a **window dispatcher** that determines the window's default behavior.

Most incoming events traverse at least five points where they can be intercepted on their journey from User Input Services to a target HI object:

4. A handler associated with the process dispatcher.
5. A handler associated with the window dispatcher.
6. A method of an `HIWindow` object.
7. A method of an `HIRootPanel` object.
8. A method of a subpanel embedded in the root panel.

In many cases the HI object containment hierarchy traversed by events extends through additional nested subpanels.

A handler associated with an Apple event dispatcher can handle an event and not pass it on; handle it and pass it on; or pass it on without taking any other action. Typically, process dispatcher handlers forward events to window dispatchers, and window dispatcher handlers call `HIWindow` methods. However, a handler associated with either kind of dispatcher may also respond to an event by sending a higher-level event to the original process dispatcher or, potentially, to any other dispatcher.

Similarly, a method of a window object, a root panel, or any other panel in the HI object containment hierarchy can handle an event and not pass it on; handle it and pass it on to a contained panel; or pass it on to a contained panel without taking any other action. Typically, window methods pass on events to the equivalent root panel methods, which pass them on to the equivalent subpanel methods, and so on until the event reaches its destination. However, like handlers associated with dispatchers, any method may also respond to an event by sending a higher-level event to the original process dispatcher, or, potentially, to any other dispatcher.

The Toolbox provides default handlers and defines `HIObject` methods for a variety of events, including both events that it defines (such as events sent by the Scrap Manager, Clipboard Manager, and Drag Manager) and events (such as mouse events or text events) sent by other parts of the system. In general, handlers and methods that correspond to the five interception points discussed above route the event to the appropriate destination. You may selectively override this default routing at any point by installing your own handlers in the appropriate dispatcher or by defining your own `HIObject` subclass and overriding the appropriate methods.

The Toolbox uses its default handlers and methods to route events in three ways:

- To a single target HI object at a specified location within a window.
- To all the HI objects in one or more windows.
- To the HI object that currently has user input focus.

In all three cases, the events are routed from a process dispatcher to a window dispatcher, from a window dispatcher to a window, from a window to a root panel, from a root panel to one or more subpanels, if necessary from subpanels to other subpanels that they in turn contain, and so on until the events reach their destinations. This standard routing for all events through the HI object containment hierarchy ensures that all embedding panels can control all aspects of the subpanels that they contain.

Events related to an application's human interface can in turn be classified according to their routing type:

- **Geometric events** are routed to a single target HI object whose bounding rectangle contains coordinates specified by the event. For example, a Mouse Down event is typically routed through the containment hierarchy to a single HI object, such as a button, that the user has clicked.
- **Broadcast events** are routed to a group of related HI objects. For example, a Window Activated event triggers the `HandleActivate` method for all the panels in a window's containment hierarchy.
- **Focused events** are routed to the HI object that currently has user input focus. For example, text events are typically routed through the containment hierarchy for HI objects with user input focus to the editable text panel currently receiving user input.

This chapter introduces each kind of routing and the advantages and disadvantages of overriding handlers or methods at each interception point.

For preliminary descriptions of some of the default handlers provided by the Toolbox, see "Standard Events Handled by the Toolbox" (page 3-5). For descriptions of event-handling methods defined by `HIObject`, see "HIObject Class Reference" (page 4-5). Documentation for subclasses of `HIObject` will be available with later developer releases.

Geometric Event Routing

In general, the event handlers installed in the process dispatcher simply forward geometric events to the dispatcher for whatever window the event occurred in. The window dispatcher in turn calls the corresponding `HIWindow` method, such as `HandleMouseUp` or `HandleMouseDown`, or the catchall method `HandleAppleEvent` if the event has no corresponding `HIWindow` method.

For information about using `HandleAppleEvent` to handle custom geometric events you define for your application, see "Routing Events With Application Handlers" (page 2-23).

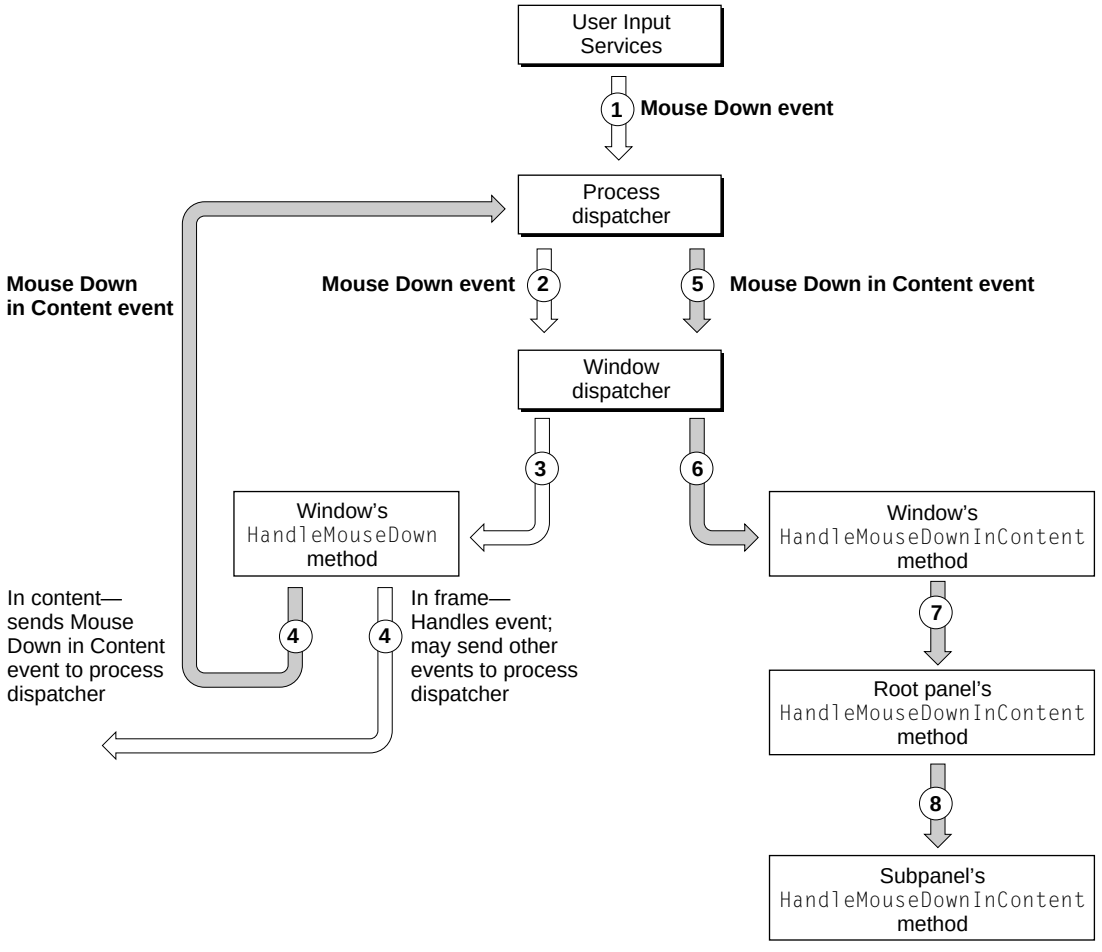
Default Geometric Event Routing

The Mouse Down event presents an interesting example of geometric event routing, because it gets translated into higher-level events, such as Mouse Down in Content or Window Resized, depending on where it occurs within a window. This illustrates the ability of the Mac OS 8 event model to interpret low-level events and resend them as higher-level synthetic or semantic events that are more meaningful to the application.

Figure 2-1 shows how the Toolbox routes a Mouse Down event to a panel in the content area of a window. These steps correspond to the numbers in the figure:

1. User Input Services determines which process should receive the Mouse Down event and sends the event to the dispatcher for that process.
2. The process dispatcher's default handler table's Mouse Down handler determines which window the event occurred in and sends a Mouse Down event to that window's dispatcher.
3. The window dispatcher's default handler table's Mouse Down handler calls `HIWindow::HandleMouseDown`.

Figure 2-1 Default geometric event routing for a Mouse Down event



4. The `HandleMouseDown` method determines whether the Mouse Down event occurred in the window's content area or in the window frame. If the event occurred in the window's content area, `HandleMouseDown` sends a Mouse Down in Content event to the process dispatcher (as shown in the figure). If the event occurred in the window frame, `HandleMouseDown` handles the event, if possible (for example, by moving or resizing the window). `HandleMouseDown` may also send additional events, such as Window Resized

or Window Close Request, to the process dispatcher to inform the application what has happened. For example, if the user attempts to close a window without saving changes, an application can respond to the Window Close Request event by displaying a dialog box that provides the opportunity to save changes before the application actually closes the window.

5. The process dispatcher's default handler table's Mouse Down in Content handler forwards the Mouse Down in Content event to the window dispatcher.
6. The window dispatcher's default handler table's Mouse Down in Content handler calls `HIWindow::HandleMouseDownInContent`.
7. The window's `HandleMouseDownInContent` method calls `HIRootPanel::HandleMouseDownInContent`.
8. The root panel's `HandleMouseDownInContent` method determines which subpanel the event occurred in and calls that subpanel's `HandleMouseDownInContent` method, which if necessary calls its subpanel's `HandleMouseDownInContent` method, and so on until the event is handled.

Overriding the Default Geometric Event Routing

You can override the default Toolbox routing for a Mouse Down event at any of the numbered steps in Figure 2-1, depending on the needs of your application:

1. A Mouse Down handler installed by the application in the process dispatcher's handler table stack can intercept the original event. The application's handler can use `FindWindow` and other static functions provided by `HIWindow` to arbitrate the recipient. This is analogous to the way mouse events are handled in System 7. Intercepting standard events at this point is not recommended for most applications.
2. A Mouse Down handler installed by the application in the window dispatcher's handler table stack can intercept the event. The application's handler can control any window behavior triggered by the event. However, if the handler returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing from this point on. If your application needs to control all aspects of event handling in a window, install your handler at this point.

3. A subclass of `HIWindow` can override the `HandleMouseDown` method. If your implementation calls the inherited `HandleMouseDown` method, you can take advantage of the default Toolbox event routing from this point on. If you wish to customize the way the window responds to events in its frame, override `HandleMouseDown` at this point.
4. A Mouse Down in Content handler installed by the application in the process dispatcher's handler table stack can intercept the event. The application's handler can use static functions provided by `HIWindow` to arbitrate the receiving panel; or, if it returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing from this point on. Intercepting standard events at this point is not recommended for most applications.
5. A Mouse Down in Content handler installed by the application in the window dispatcher's handler table stack can intercept the event. The application's handler can control any behavior in the window's content triggered by the event. However, if the handler returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing from this point on. This is one point at which you can implement your own application content.
6. A subclass of `HIWindow` can override the `HandleMouseDownInContent` method. If your implementation calls the inherited `HandleMouseDownInContent` method, you can take advantage of the default Toolbox event routing from this point on. This is another point at which you can implement your own application content.
7. A subclass of `HIRootPanel` can override the `HandleMouseDownInContent` method. If your implementation calls the inherited `HandleMouseDownInContent` method, you can take advantage of the default Toolbox event routing from this point on. However, in most cases the root panel just passes the event to one of its subpanels. Implementing content at this point is not recommended for most applications.
8. Any other subclass of `HIObject` can override the `HandleMouseDownInContent` method. If your implementation calls the inherited `HandleMouseDownInContent` method, you can take advantage of the default Toolbox event routing from this point on. This is the point at which you can customize the standard HI objects defined by the Toolbox.

You can override other standard geometric events in an analogous manner.

Note
Although intercepting standard events at the process dispatcher level is not recommended for most situations, it may sometimes be useful for a subclass of `HIObject` to override events at this level temporarily to enforce a modal state. For example, the implementation of any HI object that needs to track mouse movement while the mouse is down should install handlers for some mouse events at this level. See the description of the `HandleMouseDownInContent` method (page 4-79) for details.

Broadcast Event Routing

Broadcast events are sent to multiple HI objects within a given scope, such as all the windows in an application or all the panels in a window.

For a broadcast event that targets all the panels in a single window, the handler installed in the process dispatcher forwards the event to that window's dispatcher. The window dispatcher in turn calls the corresponding `HIWindow` method, such as `HandleActivate`, or the catchall method `HandleAppleEvent` if the event has no corresponding `HIWindow` method. The `HIWindow` method for a broadcast event in turn calls the equivalent `HIRootPanel` method, which calls the same method on all its embedded subpanels, and so on until the event has reached all panels in the container hierarchy.

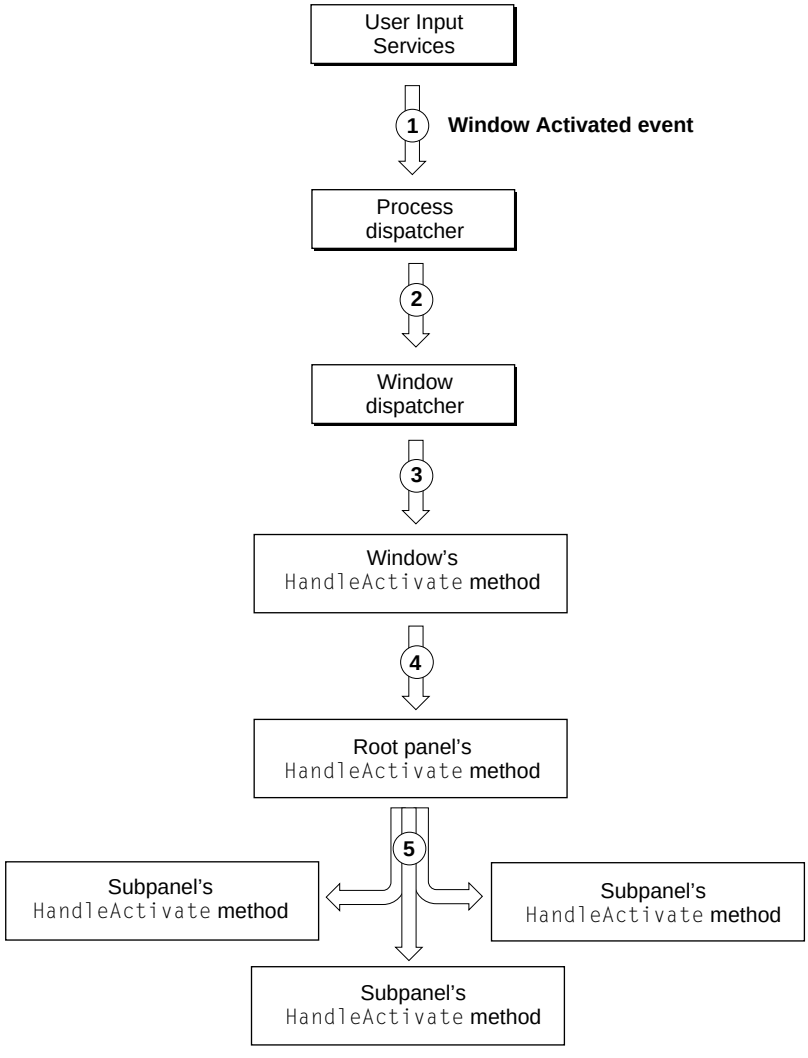
For more information about using `HandleAppleEvent` to handle custom broadcast events, see "Routing Events With Application Handlers" (page 2-23).

Default Broadcast Event Routing

Figure 2-2 shows how the Toolbox routes a Window Activated event to all the panels in a window. These steps correspond to the numbers in the figure:

1. User Input Services determines which process should receive the Window Activated event and sends the event to the dispatcher for that process.
2. The process dispatcher's default handler table's Window Activated handler determines which window the event occurred in and sends a Window Activated event to that window's dispatcher.

Figure 2-2 Default broadcast event routing for a Window Activated event



3. The window dispatcher's default handler table's Window Activated handler calls `HIWindow::HandleActivate`.
4. The window's `HandleActivate` method calls `HIRootPanel::HandleActivate`.

5. The root panel's `HandleActivate` method calls the `HandleActivate` methods for all of its embedded subpanels, which if necessary call the `HandleActivate` methods for their embedded subpanels, and so on until the event has reached all panels in the container hierarchy.

Overriding Default Broadcast Routing

You can override the default Toolbox routing for a Window Activated event at any of the numbered steps in Figure 2-1, depending on the needs of your application:

1. A Window Activated handler installed by the application in the process dispatcher's handler table stack can intercept the original event. This is analogous to the way similar events are handled in System 7. Intercepting broadcast events at this point isn't recommended for most applications. However, if the handler returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing from this point on.
2. A Window Activated handler installed by the application in the window dispatcher's handler table stack can intercept the event. If the handler returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing from this point on. You should install a handler at this point if you want to implement your own window activation without using SOM to create your own `HIWindow` subclass.
3. A subclass of `HIWindow` can override the `HandleActivate` method. If your implementation calls the inherited `HandleActivate` method, you can take advantage of the default Toolbox event routing from this point on. This is usually the best point at which to implement your own window activation.
4. A subclass of `HIRootPanel` can override the `HandleActivate` method. If your implementation calls the inherited `HandleActivate` method, you can take advantage of the default Toolbox event routing from this point on. In most cases the root panel just passes the event on to its subpanels. Subclassing `HIRootPanel` isn't recommended for implementing window content.
5. Any other subclass of `HIObject` can override the `HandleActivate` method. If your implementation calls the inherited `HandleActivate` method, you can take advantage of the default Toolbox event routing from this point on. This is the point at which you can customize the standard HI objects.

You can override other standard broadcast events in an analogous manner.

Focused Event Routing

The ultimate destination of a focused event is the HI object that currently has user input focus. Focused events typically convey user input from a keyboard. In the case of the default focused event routing supported by the Toolbox, the event that actually reaches a particular HI object represents the end product of several levels of processing performed by User Input Services and the Text Services Manager and its services, such as input methods, spelling checkers, and other language-sensitive services.

For example, User Input Services translates input from a keyboard into three kinds of events:

- **Command events.** When the user presses the Command key and at the same time presses another key, User Input Services sends the process dispatcher for the active application a Command event.
- **Navigation events.** When the user presses certain keys used to provide alternate access to HI objects (for example, the Tab key or the space bar), User Input Services sends the process dispatcher a Navigation event. If an HI object with user input focus receives the event, it responds with an appropriate action, such as transferring user input focus to the next panel that can take focus or selecting a checkbox. If a Navigation event isn't handled and if it is an appropriate key event, it gets resent to the process dispatchers a Virtual Key event.
- **Virtual Key events.** When the user presses any key other than a Command key combination or a key used for navigation, User Input Services sends the process dispatcher for the active application a Virtual Key event.

Virtual Key events convey individual character codes. Handlers installed in the process dispatcher use the Text Services Manager to convert Virtual Key events into high-level text events. For example, certain combinations of Virtual Key events might be translated into a ligature, a Kanji character, or a mathematical symbol, depending on text input methods and other services currently registered with the Text Services Manager.

Your application doesn't normally handle a Virtual Key event directly. Instead, you can either take advantage of the default handling or override handlers for the high-level text events.

In addition to creating text events from Virtual Key events, the Text Services Manager creates them from other sources of input, such as voice recognition software. Regardless of the way text is generated, your application always handles high-level text events the same way. This is one of the benefits of the default event routing provided by the Toolbox and the Text Services Manager.

If you need to intercept low-level keyboard events before they are translated into Command events, Navigation events, or Virtual Key events—for example, for a game that can be controlled from the keyboard—you can override handlers in the process dispatcher's default handler table for the Key Up, Key Down, and Auto Key events.

This section introduces the way the Text Services Manager and the Toolbox cooperate to process Command events, Navigation events, Virtual Key events, and text events after they arrive at the process dispatcher. For an introduction to the treatment of text in Mac OS 8, see the accompanying document *Text Handling and Internationalization.* Documentation for the Text Services Manager will be available with later developer releases.

Command Events

User Input Services generates Command events when the user triggers a shortcut for some action, for example by pressing the Command key and the O key at the same time as a shortcut for the Open command in the File menu. However, Command events aren't restricted to keyboard equivalents for menu items. User Input Services may translate other forms of user input, such as speech, into Command events, and Command events may be used to trigger actions not represented by menu items.

Unlike Virtual Key events and Navigation events, Command events are never intercepted by the Text Services Manager.

Future developer releases will provide information about Command events.

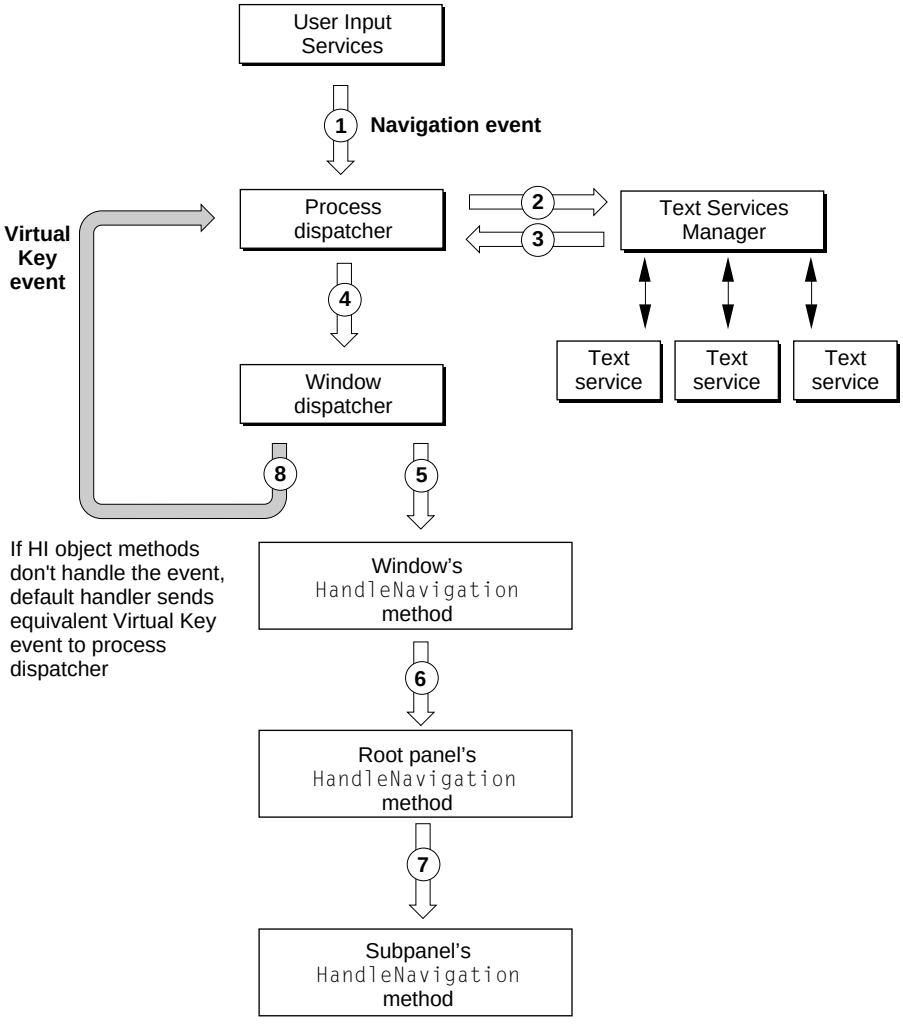
Navigation Events

User Input Services generates Navigation events when the user takes some action that can be interpreted as a navigation request; for example, the user may press the Tab key to transfer user input focus from one HI object to another in a dialog box. However, Navigation events aren't restricted to keyboard input. User Input Services may translate input from other sources of user input, such as speech recognition software, into Navigation events.

Default Routing for a Navigation Event

Figure 2-3 shows the default routing for a typical Navigation event from User Input Services to the panel that currently has user input focus.

Figure 2-3 Default focused event routing for a Navigation event



These steps correspond to the numbers in Figure 2-3:

1. User Input Services sends the event to the process dispatcher for the active application.
2. The process dispatcher's default handler table's Navigation event handler first passes the event to the Text Services Manager, which gives each of its currently registered services (such as input methods, spelling checkers, and so on) a chance to handle the event. If one of the text services handles the Navigation event, the event goes no further.
3. If none of the text services handles the event, the Text Services Manager returns the event to the Navigation event handler.
4. The Navigation event handler forwards the event to the window that currently has user input focus.
5. The window dispatcher's default handler table's Navigation event handler calls `HiWindow::HandleNavigation`.
6. The window's `HandleNavigation` method calls `HIRootPanel::HandleNavigation`.
7. The root panel's `HandleNavigation` method calls the `HandleNavigation` method of the subpanel that currently has user input focus, which if necessary calls the `HandleNavigation` method of its subpanel, and so on until the event is handled.
8. If the event isn't handled by any of the method calls, the call to `HIWindow::HandleNavigation` (step 5) returns `errAEEventNotHandled` and the window dispatcher's default handler resends the event as a Virtual Key event.

For example, if the user presses the Tab key, User Input Services sends the process dispatcher a Navigation event that specifies a Tab key was pressed (step 1). The default Navigation event handler passes the event to the Text Services Manager (step 2), which checks whether any of its services can handle the event—for example, to move the cursor from one part of a spelling checker to another. If the Navigation event is handled by a text service, it proceeds no further.

If the Navigation event isn't handled by a text service, the Text Services Manager returns the event to the Navigation event handler (step 3), which forwards it to the appropriate window dispatcher (step 4). The window dispatcher's default handler then calls the window's `HandleNavigation` method (step 5), and so on through the containment hierarchy for panels with user input focus until the event gets handled, for example by dialog box transferring user input focus to the next focusable subpanel.

If the window that currently has user input focus can't handle the Tab keypress as a Navigation event, the window may be able to handle it in the form of a key event—for example, as a Tab character within a text panel or a text document. Therefore, if the panels to which the window dispatcher's default handler passes the Navigation event don't handle it, and if the event is an appropriate key event, the handler resends the event to the process dispatcher as a Virtual Key event (step 8). This allows a keypress a chance to be processed as text input rather than navigation input.

Overriding the Default Routing for a Navigation Event

You can override the default Toolbox routing for a Navigation event at most of the steps shown in Figure 2-3, depending on the needs of your application:

1. A Navigation event handler installed by your application in the process dispatcher's handler table stack can intercept the original Navigation event. Intercepting Navigation events at this point isn't recommended for most applications.
2. If a Navigation event handler associated with the process dispatcher returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing—including the default handler's use of the Text Services Manager—from this point on.
3. You can't override the behavior of the Text Services Manager after it receives a Navigation event. If none of its currently registered services handles the event, the Text Services Manager returns the event to the Navigation event handler that called it.

4. A Navigation event handler installed by your application in the window dispatcher's handler table stack can intercept the event. If the handler itself returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing from this point on. You should install a handler at this point if you want to implement your own Navigation event handling without using SOM to create your own `HIWindow` subclass.
5. A subclass of `HIWindow` can override the `HandleNavigation` method. If your implementation calls the inherited `HandleNavigation` method, it can take advantage of the default Toolbox event routing from this point on. This is usually the best point at which to implement custom navigation within a window.
6. A subclass of `HIRootPanel` can override the `HandleNavigation` method. If your implementation calls the inherited `HandleNavigation` method, it can take advantage of the default Toolbox event routing from this point on. However, in most cases the root panel just forwards the event to one of its subpanels. Subclassing `HIRootPanel` isn't recommended for implementing focused events.
7. Any other subclass of `HIObject` can override the `HandleNavigation` method. If your implementation calls the inherited `HandleNavigation` method, you can take advantage of the default Toolbox event routing from this point on. This is the point at which you can customize the navigation behavior of the standard HI objects defined by the Toolbox.
8. If your application installs a Navigation event handler in the window dispatcher's handler table stack, the handler should resend the event to the process dispatcher as a Virtual Key event if the original Navigation event isn't handled by the window or any of its subpanels.

Virtual Key Events and Text Events

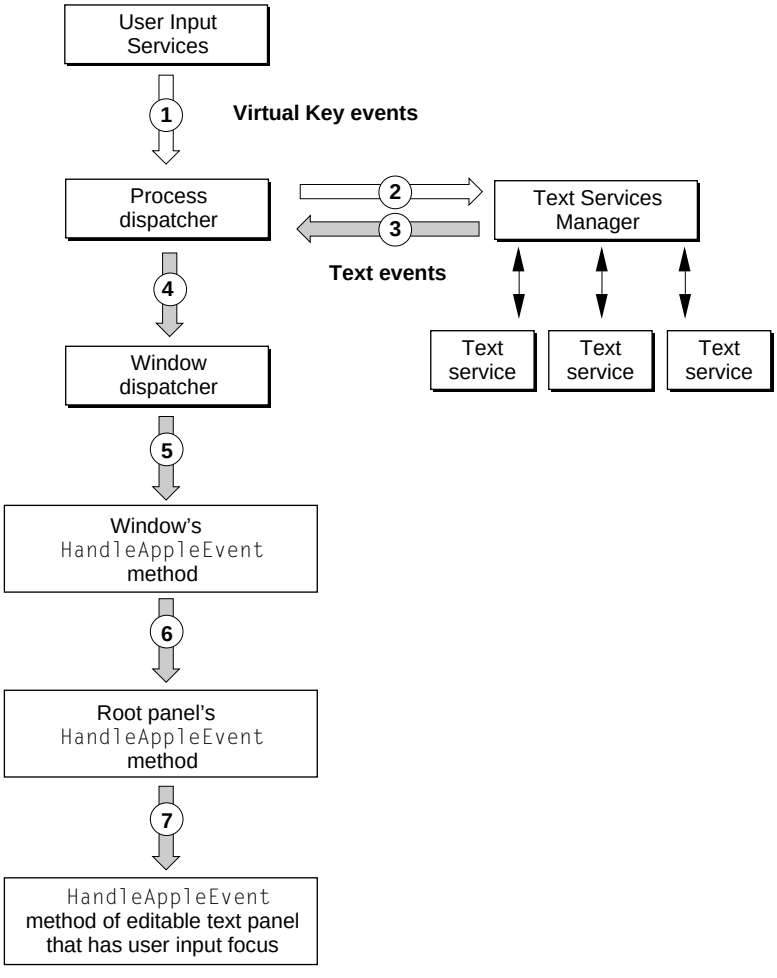
The Text Services Manager intercepts all Virtual Key events and transforms them into a series of text events with the aid of input methods and other text services. The Text Services Manager expresses the results of this processing by means of four kinds of text events that it sends to the process dispatcher:

- **Update Active Input Area events.** The Text Services Manager uses Update Active Input Area events to convey one or more textual characters in the form of a text object, plus related information about which portions of existing text content to replace and how to highlight the new text.
- **Position to Offset events.** The Text Services Manager uses Position to Offset events to request that the HI object with user input focus translate a global coordinate position to an offset within the object's textual content. Position to Offset events allow text services to determine actions associated with mouse movement, text selection, and so on.
- **Offset to Position events.** The Text Services Manager uses Offset to Position events to request that the HI object with user input focus translate an offset within the object's textual content to a global coordinate position. Offset to Position events allow text services to obtain information about the attributes of a particular portion of text within the HI object's textual content and the absolute position of the text on the screen.
- **Get Input Area Region events.** The Text Services Manager uses Get Region events to request ranges of text within an HI object's textual content. This event allows text services to manipulate ranges of text; for example, it allows an interactive spell checker to manipulate existing text.

Default Routing for Virtual Key and Text Events

Figure 2-4 shows the default routing for all Virtual Key events and for the text events into which the Text Services Manger transforms Virtual Key events.

Figure 2-4 Default focused event routing for Virtual Key and Text events



These steps correspond to the numbers in Figure 2-4:

1. User Input Services (or potentially other sources, such as the window dispatcher's default handler for a Navigation event) sends Virtual key events to the process dispatcher for the active application.
2. The process dispatcher's default handler table's Virtual Key event handler passes the event to the Text Services Manager, which translates the incoming stream of Virtual Key events into text events.
3. The Text Services Manager sends the text events back to the process dispatcher.
4. The process dispatcher's default handler table's text event handlers forward the events to the window that currently has user input focus.
5. The window dispatcher's default handler table's text event handlers call `HIWindow::HandleAppleEvent`.
6. The window's `HandleAppleEvent` method calls `HIRootPanel::HandleAppleEvent`.
7. The root panel's `HandleAppleEvent` method calls the `HandleNavigation` method of the subpanel with user input focus, which if necessary calls the `HandleAppleEvent` method of its subpanel with user input focus, and so on until the event is handled.

Overriding Default Routing for Virtual Key and Text Events

You can override the default Toolbox routing for Virtual Key events at most of the numbered steps shown in Figure 2-4, depending on the needs of your application:

1. Virtual Key event handlers installed by your application in the process dispatcher's handler table stack can intercept the original Virtual Key events. Intercepting Virtual Key events at this point isn't recommended for most applications. If you want to receive low-level keypress information before it is translated into Virtual Key events, you can install your own Key Up, Key Down, and Auto Key event handlers in the process dispatcher's default handler table.
2. If a Virtual Key event handler associated with the process dispatcher returns `errAEEventNotHandled`, it can take advantage of the default Toolbox event routing—including the default handler's use of the Text Services Manager—from this point on.

3. You can't override the behavior of the Text Services Manager after it receives Virtual Key events.
4. Text event handlers installed by your application in the window dispatcher's handler table stack can intercept the text events generated by the Text Services Manager. If such handlers returns `errAEEventNotHandled`, they can take advantage of the default Toolbox event routing from this point on. You should install handlers at this point if you want to implement your own text event handling without using SOM to create your own `HIWindow` subclass.
5. A subclass of `HIWindow` can override the `HandleAppleEvent` method. If your implementation calls the inherited `HandleAppleEvent` method, it can take advantage of the default Toolbox event routing from this point on. This is usually the best point at which to implement custom navigation within a window.
6. A subclass of `HIRootPanel` can override the `HandleAppleEvent` method. If your implementation calls the inherited `HandleAppleEvent` method, it can take advantage of the default Toolbox event routing from this point on. However, in most cases the root panel just forwards the event to one of its subpanels. Subclassing `HIRootPanel` isn't recommended for implementing focused events.
7. Any other subclass of `HIObject` can override the `HandleAppleEvent` method. If your implementation calls the inherited `HandleAppleEvent` method, it can take advantage of the default Toolbox event routing from this point on. This is the point at which you can customize the text-handling behavior of the standard editable text panel defined by the Toolbox.

If you override `HandleAppleEvent` in any `HIPanel` subclass, the panel's initialization methods should register the panel's interest in the events you want it to handle by calling its `RegisterInterestInEvent` method. For more information about registering interest in events, see "Registering a Panel's Interest in an Event" (page 2-24).

Routing Events With Application Handlers

As discussed in the preceding sections, you can override the default Toolbox handlers for the standard events by installing your own handlers in the appropriate dispatcher or by subclassing from any HI object class. In a similar manner, you can install handlers for any other events in a process dispatcher or a window dispatcher, and you can subclass any panel class to implement custom event handling.

Handler Tables in Process Dispatchers

Overriding at the process level gives you the most control over event routing in your application, but it also requires you to do the most work. In general, overriding the Toolbox handlers at this level requires you to implement your application in much the same way as a System 7 application—that is, by controlling event dispatching within the application without any assistance from the Toolbox. Overriding at this level also has the greatest potential to limit your application's compatibility with future versions of the Mac OS. Because you can use little if any of the default Toolbox event-handling code if you override at this level, you also limit the extent to which your application can automatically take advantage of new events and event handling supported by the Mac OS in the future.

If your application structure requires you to override at this level, your handlers should first determine if the event is relevant to one of your windows. If it is not, your handler should simply return `errAEEventNotHandled` and let the default Toolbox event routing handle the event. This can be useful, for example, if you are attempting to reuse System 7 code in a Mac OS 8 application and you want to take advantage of the default Toolbox event routing for any events that your application doesn't need to intercept, such as a mouse event in an Apple Guide window.

Handler Tables in Window Dispatchers

Overriding at the window level, either via procedural handlers installed in the window dispatcher or subclasses of `HIWindow`, is the recommended practice for all Mac OS 8 savvy applications. Overriding at this level allows the Toolbox to continue to perform default event routing, so that your application never sees events that it need not handle—for example, events targeted at system floating windows such as Text Services input method windows or utility windows created by other background services.

Registering a Panel's Interest in an Event

To create a subclass for panels that need to handle events other than the standard Toolbox events, you must override the `HandleAppleEvent` method and handle the events from within your subclass implementation.

The initializing methods for your subclass should register the panel's interest in any event you want it to handle by calling its `RegisterInterestInEvent` method. In addition to the event class and ID, you pass a handler table reference and a routing type to `RegisterInterestInEvent`. The handler table reference identifies the handler table with which you want to register the panel's interest.

`RegisterInterestInEvent` installs a handler that routes the event through the HI object containment hierarchy according to its routing type. This ensures that the event will be passed to the `HandleAppleEvent` methods for the appropriate window, root panel, and subpanels until it either reaches a panel that can handle it or reaches a panel whose `HandleAppleEvent` method returns `errAEEventNotHandled`.

If you specify the event's routing type as geometric (`kHIRouteByLocation`), it gets sent through the containment hierarchy to the panel, if any, in the location where the event occurred. If you specify the event's routing type as focused (`kHIRouteToFocusSubPanel`), it gets sent through the containment hierarchy for HI objects with user input focus to the panel that currently has user input focus. If you specify the event's routing type as broadcast (`kHIRouteToAllSubPanels`), the root panel sends it to all subpanels.

Because events of all routing types are always routed through a containment hierarchy, embedding panel implementations of the `HandleAppleEvent` method should call the `HandleAppleEvent` method of its subpanels, if possible, according to the event's routing type. The last panel to receive the event, at the bottom of the containment hierarchy, should return `errAEEventNotHandled` if it can't handle the event.

Toolbox Support for Modal States

To implement a modal state, you typically create a filtered handler table and install handlers in it for the events to which you want to limit event handling. What you do next depends on whether the modal state is associated with a dialog box or alert box:

- If you are implementing a modal state that doesn't involve a dialog box or alert box, you must use Apple Event Manager functions to push the table onto a dispatcher's handler stack (`AEPushDispatcherHandlerTable`) and begin receiving events (`AEReceive`). When the user dismisses the dialog box, you pop the handler table off the handler stack (`AEPopDispatcherHandlerTable`).
- If you are using a standard modal dialog box or alert box provided by the HI Objects class library, you call a single method (`ExecuteModality`) that performs all the same operations automatically.

For more information about using the Apple Event Manager to implement event dispatching for modal states, see the accompanying document *Apple Events in Mac OS 8*.

More information about Toolbox support for modal states will be provided with later developer releases.

Toolbox Events Reference

Contents

This chapter introduces some of the standard Apple events for which the Toolbox provides handlers, including the parameters of each event, the descriptor types it uses, and the behavior of the default handlers.

Before you read this chapter, you should be familiar with the accompanying document *Apple Events in Mac OS 8* and with the first two chapters in this document: "Introduction to the Mac OS 8 Toolbox" (page 1-3) and "Toolbox Event Routing" (page 2-3).

▲ **WARNING**

This document is preliminary and incomplete. It is intended only to illustrate the design concepts that underlie both the standard events and the default handling provided by the Toolbox. All information presented here, including details such as event class and event ID and the behavior of individual handlers, is subject to change. ▲

Apple Event Descriptor Types

Table 3-1 lists some of the descriptor types defined by the Apple Event Manager for use with the Apple events defined in this chapter. Information about additional descriptor types, including those used with text events, will be provided with later developer releases.

Note
Table 3-1 doesn't include descriptor types used with object specifier records or specialized types used by the System 7 Apple Event Manager. Mac OS 8 savvy applications can use most descriptor types defined by the System 7 Apple Event Manager, as documented in *Inside Macintosh: Interapplication Communication.* ◆

Table 3-1 Descriptor types defined by the Apple Event Manager for use with the standard events

Descriptor type	Value	Description
`typeBoolean`	`'bool'`	1-byte Boolean value
`typeChar`	`'TEXT'`	Unterminated string
`typeLongInteger`	`'long'`	32-bit integer
`typeShortInteger`	`'shor'`	16-bit integer
`typeTrue`	`'true'`	Boolean value true
`typeFalse`	`'fals'`	Boolean value false
`typeAlias`	`'alis'`	Alias record
`typeEnumerated`	`'enum'`	Enumerated data
`typeType`	`'type'`	Four-character code for event class or event ID
`typeQDPoint`	`'qdpt'`	QuickDraw point
`typeQDRectangle`	`'qdrt'`	QuickDraw rectangle
`typeHIWindow`	`'wobj'`	A pointer to a window object
`typeHIMenu`	`'mobj'`	A pointer to a menu object

Standard Events Handled by the Toolbox

Key Events

The Key Down, Auto Key, and Key Up events are low-level keypress events generated by a keyboard family. Most applications aren't interested in these events; instead, they are interested in higher-level translations.

Default handlers provided by the Toolbox use User Input Services to translate key events into Command, Navigation, or Virtual Key events. For an introduction to the way the Toolbox routes these events, see "Toolbox Event Routing" (page 2-3). Later developer releases will provide more information about these events.

Key Down

Indicates that a particular key has been pressed.

Event class	`kAEKeyClass`	
Event ID	`kAEDown`	
Parameters—		
`keyModifiers`	`typeShortInteger`	Key codes for modifier keys held down while the key was pressed.
`keyWhen`	`typeLongInteger`	Time that the key was pressed.
`keyWhere`	`typeQDPoint`	The location of the pointer at the time the key was pressed.
`keyKey`	`typeChar`	Key pressed.
`keyKeyCode`	`typeChar`	Character code for the key pressed.
`keyKeyboard`	`typeChar`	Keyboard on which the key was pressed.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Key Down handler associated with the default process dispatcher uses User Input Services to combine the event with other incoming key events as necessary to form Command, Navigation, and Virtual Key events.

Auto Key

Indicates that a particular key has been held down.

Event class `kAEKeyClass`

Event ID `kAEAutoDown`

Parameters—

`keyModifiers`	`typeShortInteger`	Key codes for modifier keys held down while the key was held down.
`keyWhen`	`typeLongInteger`	Time that the key was held down.
`keyWhere`	`typeQDPoint`	The location of the pointer at the time the key was held down.
`keyKey`	`typeChar`	Key held down.
`keyKeyCode`	`typeChar`	Character code corresponding to the key that was held down.
`keyKeyboard`	`typeChar`	Keyboard on which the key was held down.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Auto Key handler associated with the default process dispatcher uses User Input Services to combine the event with other incoming key events as necessary to form Command, Navigation, and Virtual Key events.

Key Up

Indicates that a particular key has been released.

Event class `kAEKeyClass`

Event ID `kAEUp`

Parameters—

`keyModifiers`	`typeShortInteger`	Key codes for modifier keys held down while the key was released.
`keyWhen`	`typeLongInteger`	Time that the key was released.
`keyWhere`	`typeQDPoint`	The location of the pointer at the time the key was released.
`keyKey`	`typeChar`	Key released.
`keyKeyCode`	`typeChar`	Character code corresponding to the key that was released.
`keyKeyboard`	`typeChar`	Keyboard on which the key was released.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Key Up handler associated with the default process dispatcher uses User Input Services to combine the event with other incoming key events as necessary to form Command, Navigation, and Virtual Key events.

Mouse Events

The default handlers installed in the process dispatcher simply forward Mouse Down, Mouse Moved, Mouse Stopped Moving, and Mouse Up events to the dispatcher for whatever window the event occurred in. The window dispatcher in turn forwards the event to the corresponding `HIWindow` method, such as `HandleMouseUp` or `HandleMouseDown`, or to the catchall method `HandleAppleEvent` if the event has no corresponding `HIWindow` method.

Mouse Up

The mouse button has been released in either the frame or the content area of a window.

Event class	`kAEMouseClass`	
Event ID	`kAEUp`	
Parameters—		
`keyModifiers`	`typeShortInteger`	Codes for modifier keys held down when the user releases the mouse button.
`keyWhere`	`typeQDPoint`	Global coordinates for the location of the pointer onscreen when the user releases the mouse button.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Mouse Up handler associated with the default process dispatcher determines which window the event occurred in and resends the event to that window's dispatcher.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Mouse Up handler associated with the window dispatcher calls `HIWindow::HandleMouseUp`.

Mouse Down

The mouse button has been pressed in either the frame or the content area of an active window.

Event class	`kAEMouseClass`	
Event ID	`kAEDown`	
Parameters—		
`keyModifiers`	`typeShortInteger`	Codes for modifier keys held down when the user presses the mouse button.
`keyWhere`	`typeQDPoint`	Global coordinates for the location of the pointer onscreen when the user presses the mouse button.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Mouse Down handler associated with the default process dispatcher determines which window the event occurred in and forwards the event to that window's dispatcher.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Mouse Down handler associated with the window dispatcher calls `HIWindow::HandleMouseDown`.

Mouse Moved

The mouse has moved in either the frame or the content area of a window.

Event class	`kAEMouseClass`	
Event ID	`kAEMoved`	
Parameters—		
`keyModifiers`	`typeShortInteger`	Codes for modifier keys held down when the user moved the mouse.
`keyWhere`	`typeQDPoint`	Global coordinates for the location of the pointer onscreen when the user moved the mouse.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Mouse Moved handler associated with the default process dispatcher determines which window the event occurred in and forwards the event to that window's dispatcher.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Mouse Moved handler associated with the window dispatcher calls `HIWindow::HandleMouseMoved`.

Mouse Stopped Moving

The mouse has stopped moving in either the frame or the content area of a window.

Event class	`kAEMouseClass`	
Event ID	`kAEStoppedMoving`	
Parameters—		
`keyModifiers`	`typeShortInteger`	Codes for modifier keys held down when the user stopped moving the mouse.
`keyWhere`	`typeQDPoint`	Global coordinates for the location of the pointer onscreen when the user stopped moving the mouse.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Mouse Stopped Moving handler associated with the default process dispatcher determines which window the event occurred in and forwards the event to that window's dispatcher.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Mouse Stopped Moving handler associated with the window dispatcher calls `HIWindow::HandleMouseStoppedMoving`.

Window Events

Mouse Down in Back

The mouse button has been pressed in either the frame or the content area of an inactive window.

Event class	`kAEWindowClass`	
Event ID	`kAEMouseDown`	
Parameters—		
`keyDirectObject`	`typeHIWindow`	A pointer to the window object receiving the event.
`keyModifiers`	`typeShortInteger`	Codes for modifier keys held down when the user pressed the mouse button.
`keyWhere`	`typeQDPoint`	Global coordinates for the location of the pointer onscreen when the user pressed the mouse button.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Mouse Down in Back handler associated with the default process dispatcher determines which window the event occurred in and forwards the event to that window's dispatcher.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Mouse Down in Back handler associated with the window dispatcher forwards the event to `HIWindow::HandleMouseDownInBack`.

Mouse Down in Content

The mouse button has been pressed in the content area of an active window.

Event class `kAEWindowClass`

Event ID `kAEMouseDown`

Parameters—

`keyDirectObject`	`typeHIWindow`	A pointer to the window object receiving the event.
`keyModifiers`	`typeShortInteger`	Codes for modifier keys held down when the user pressed the mouse button.
`keyWhere`	`typeQDPoint`	Global coordinates for the location of the pointer onscreen when the user pressed the mouse button.
`keyLocalWhere`	`typeQDPoint`	Local coordinates in the window's color graphics port for the same point described by the `keyWhere` parameter.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Mouse Down in Content handler associated with the default process dispatcher determines which window the event occurred in by examining the event's direct object. It then forwards the event to that window's dispatcher. This handler does not perform any hit testing.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Mouse Down in Content handler associated with the window dispatcher calls `HIWindow::HandleMouseDownInContent`.

Window Resized

The window has been resized.

Event class	`kAEWindowClass`	
Event ID	`kAEResized`	
Parameters—		
`keyDirectObject`	`typeHIWindow`	A pointer to the window object that has been resized.
`keyOriginalBounds`	`typeQDRectangle`	The window's original port rectangle.
`keyNewBounds`	`typeQDRectangle`	The window's new port rectangle.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Window Resized handler associated with the default process dispatcher forwards the event to the dispatcher for the window that has been resized.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Window Resized handler associated with the window dispatcher calls `HIWindow::HandleResize`.

Window Close Request

The user has indicated that a window should be closed, for example by clicking the window's close box.

Event class	`kAEWindowClass`	
Event ID	`kAEClosed`	
Parameters—		
`keyDirectObject`	`typeHIWindow`	A pointer to the window object that should be closed.
`keyCloseAllWindows`	`typeBoolean`	A Boolean value indicating whether all the application's windows should be closed. A value of `true` indicates that the application should close all its open windows; a value of `false` indicates that the client should close only the window specified by the `keyDirectObject` parameter.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Window Close Request handler associated with the default process dispatcher forwards the event to the dispatcher for the window or windows to be closed.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Window Close Request handler associated with the window dispatcher calls `HIWindow::HandleClose`.

Window Activated

The window has been activated.

Event class	`kAEWindowClass`	
Event ID	`kAEActivate`	
Parameters—		
`keyDirectObject`	`typeHIWindow`	A pointer to the window object that has been activated.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Window Activated handler associated with the default process dispatcher forwards the event to the dispatcher for the window that has been activated.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Window Activated handler associated with the window dispatcher calls `HIWindow::HandleActivate`.

Window Deactivated

The window has been deactivated.

Event class	`kAEWindowClass`	
Event ID	`kAEDeactivate`	
Parameters—		
`keyDirectObject`	`typeHIWindow`	A pointer to the window object that has been deactivated.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Window Deactivated handler associated with the default process dispatcher forwards the event to the dispatcher for the window that has been deactivated.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Window Deactivated handler associated with the window dispatcher calls `HIWindow::HandleDeactivate`.

Update

Update the window.

Event class	`kAEWindowClass`	
Event ID	`kAEDeactivate`	
Parameters—		
`keyDirectObject`	`typeHIWindow`	A pointer to the window object that requires updating.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Update handler associated with the default process dispatcher forwards the event to the dispatcher for the window to be updated.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Update handler associated with the window dispatcher forwards the event to `HIWindow::HandleUpdate`, which in turn calls `HIWindow::BeginUpdate` and `HIWindow::EndUpdate`.

Text Events

The Text Services Manager intercepts all Virtual Key events before they reach the default process dispatcher's application handler tables or default handler table. With the aid of input methods and other text services, the Text Services Manager transforms Virtual Key events into a series of text events, which the Apple Event Manager then routes through the handler table stack in the usual manner.

IMPORTANT

You don't need to know about the structure of text events unless you are implementing your own text processor or subclassing from `HIEditableText`. Otherwise, the default Toolbox text event routing and the standard editable text panels handle all text events automatically. ▲

Before attempting to use the information in this section, you should be familiar with the accompanying document *Text Handling and Internationalization* and with the System 7 book *Inside Macintosh: Text.*

Detailed documention for the Mac OS 8 Text Services Manager will be available with later developer releases.

Update Active Input Area

The text content of the active input area has been updated, highlighted, scrolled, or committed. The Update Active Input Area event conveys one or more textual characters in the form of a text object, plus related information about which portions of existing text content to replace and how to highlight the new text.

Event class `kAETextEventClass`

Event ID `kAEUpdateActiveInputArea`

Parameters—

`kAETextEventText`	`typeTextObject`	The text to be inserted.
`keyAETextInputObject`	`typeTextInputObject`	The keyboard object that was active when the text was generated.
`keyAEFixedLength`	`typeLongInteger`	The number of bytes that should be committed to the application's text content.

keyAEContext	typeTSMContext	The TSM context that generated this event.
keyAEInlineID	typeLongInteger	The inline hole ID to be allocated to this transaction.
keyAEReplaceRange	typeReplaceRangeArray	The Replace Range array, indicating the range of text in the active input area to be replaced
keyAEHiliteRange	typeHilteRangeArray	The Hilight Range array, indicating the range of text in the active input area to be highlighted.
keyAEPinRange	typeTextRange	The Pin range, indicating the range of text that should be scrolled into view if possible.
keyAEClauseOffsets	typeClauseOffsetArray	The Clause Range array, used for word selection and related purposes.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Update Active Input Area event handler associated with the default process dispatcher forwards the event to the dispatcher for the window that has user input focus.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Update Active Input Area event handler associated with the window dispatcher calls the `HandleAppleEvent` method of the editable text panel that has user input focus.

Position to Offset

Requests that the HI object with user input focus translate a global coordinate position to an offset within the object's textual content. The reply to a Position to Offset event allows text services to determine actions associated with mouse movement, text selection, and so on.

Event class	`kAETextEventClass`	
Event ID	`kAEPosToOffset`	
Parameters—		
`kAEPoint`	`typeQDPoint`	The global coordinate position to offset.
`keyAEDragging`	`typeBoolean`	If the value is `true`, the mouse is being dragged; if the value is `false`, the mouse is not being dragged.
`keyAEContext`	`typeTSMContext`	The TSM context that generated this event.
Reply parameters—		
`kAERegionClass`	`typeRegionClass`	The region the point falls in.
`keyAEOffset`	`typeByteOffset`	The requested offset within the HI object's text.
`keyAELeftSide`	`typeBoolean`	If the value is `true`, the point falls on the left side of a character. If the value is `false`, the point falls on the right side of a character.
`keyAEInlineOwner`	`typeTSMContext`	The TSM context that owns the inline hole (if the point falls within an inline hole).
`keyAEInlineID`	`typeLongInteger`	The inline hole ID (if the point falls within an inline hole).

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Position to Offset event handler associated with the default process dispatcher forwards the event to the dispatcher for the window that has user input focus.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Position To Offset event handler associated with the window dispatcher calls the `HandleAppleEvent` method of the editable text panel that has user input focus.

Offset to Position

Requests that the HI object with user input focus translate an offset within the object's textual content to a global coordinate position. The reply to an Offset to Position event allows text services to obtain information about the attributes of a particular portion of text within the HI object's textual content and the absolute position of the text on the screen.

Event class	`kAETextEventClass`	
Event ID	`kAEOffsetToPos`	
Parameters—		
`keyAEOffset`	`typeByteOffset`	The requested offset within the HI object's text.
`keyAEInlineID`	`typeLongInteger`	The inline hole ID (if the point falls on an inline hole).
`keyAEContext`	`typeTSMContext`	The TSM context that generated this event.
Reply parameter—		
`kAETextInlineInfo`	`typeTextLineInfo`	The Text Info structure describing the point in the client document.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Offset to Position event handler associated with the default process dispatcher forwards the event to the dispatcher for the window that has user input focus.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Offset to Position event handler associated with the window dispatcher calls the `HandleAppleEvent` method of the editable text panel that has user input focus.

Get Input Area Region

Requests ranges of text within an HI object's textual content. The reply to a Get Input Area Region event consists of a range of text that text services can manipulate.

Event class	`kAETextEventClass`	
Event ID	`kAEOffsetToPos`	
Parameters—		
`kAERegionClass`	`typeRegionClass`	The request region type.
`keyAEGetRegionRange`	`typeGetRange`	The region array for the region to be retrieved.
`keyAEInlineID`	`typeLongInteger`	The inline hole ID (if the requested region is within an inline hole).
`keyAEContext`	`typeTSMContext`	The TSM context that generated this event.
Reply parameter—		
`kAETextEventText`	`typeTextObject`	The text of the requested region.

DEFAULT HANDLER FOR PROCESS DISPATCHER

The default Get Input Area Region event handler associated with the default process dispatcher forwards the event to the dispatcher for the window that has user input focus.

DEFAULT HANDLER FOR WINDOW DISPATCHER

The default Get Input Area Region event handler associated with the window dispatcher calls the `HandleAppleEvent` method of the editable text panel that has user input focus.

Application Events

Standard Apple events previously defined by Apple—for example, the Required suite of Apple events discussed in *Inside Macintosh: Interapplication Communication*—are still supported in Mac OS 8 and still play the same roles. Mac OS 8 defines additional events that all applications should support, including Suspend and Resume.

Suspend

Informs an application in the foreground that the Process Manager is about to switch it into the background.

Event class `kAEApplicationClass`
Event ID `kAESuspend`
No parameters

The Toolbox doesn't provide any default handlers for the Suspend event. Your applciation should install a Suspend event handler in the default process dispatcher that deactivates the front window, removes the highlighting from any selections, and does anything else required to get ready for switching out.

Resume

Informs an application in the background that the Process Manager is about to switch it into the foreground.

Event class `kAEApplicationClass`
Event ID `kAEResume`
No parameters

The Toolbox doesn't provide any default handlers for the Resume event. Your application should install a Resume event handler in the default process dispatcher that activates the front window and restores any windows to the state the user left them at the time of the previous Suspend event.

HIObject Class Reference

Contents

HIObject

Superclass `SOMObject`

Subclasses See Figure 4-1 (page 4-6).

The abstract base class `HIObject` defines the behavior common to all HI objects.

For an introduction to the HI Objects class library, see "Introduction to the Mac OS 8 Toolbox" (page 1-3).

Description

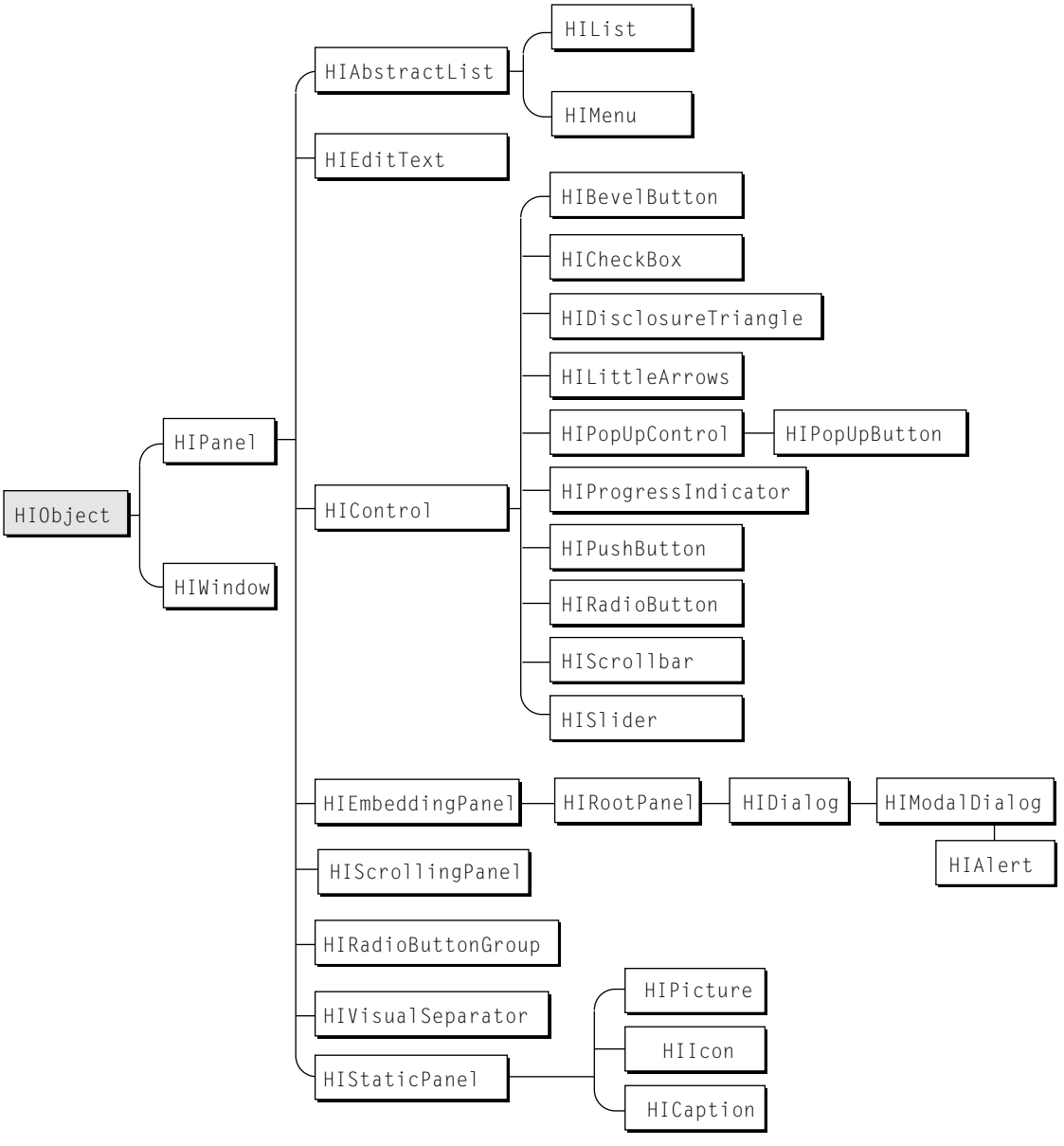
The abstract superclass `HIObject` defines constants, data types, public methods, protected methods, and static methods for use by clients of the HI Objects class library. The public methods perform operations common to all human interface objects, such as basic event handling, manipulating an object's location, enabling and disabling it, setting its visibility, controlling user input focus, imaging, and so on. The static methods (that is, procedural functions you can call without specifying a particular object) perform operations that aren't targeted to a specific human interface object, such as creating a new HI object and initializing it using data stored in a resource.

`HIObject` also defines protected methods, which you need to use only if you are implementing your own subclass based on `HIObject` or one of its subclasses.

If your application uses the HI Objects class library as a client or if you need to subclass your own HI objects, you need information about both class `HIObject` and the other standard HI object classes.

You should never instantiate `HIObject` itself. Instead, instantiate either one of the subclasses defined by Apple or a subclass defined by your application.

Figure 4-1 `HIObject` and the inheritance hierarchy for the HI Objects class library



Summary of Static Methods

For detailed descriptions of these methods, see "Static Methods" (page 4-19).

`GetNewHIObjectFromResource`
: Reads a resource into memory and uses its data to instantiate and initialize a new HI object.

`GetNewHIObject`
: Instantiates and initializes an HI object using data in an AE record.

`GetHIObjectFromRefLabel`
: Finds all the HI objects in the current process that have a specified reference label.

Summary of Public Methods

For detailed descriptions of these methods, see "Public Methods" (page 4-25).

Initializing, Saving, and Releasing

`Init`
: Initializes an existing HI object programmatically.

`InitFromAERecord`
: Initializes an HI object on the basis of data in an AE record.

`WriteToAERecord`
: Writes an HI object's state and initialization data to an AE record that can be saved on disk, sent as part of an Apple event, or passed to `InitFromAERecord`.

`Clone`
: Returns a new pointer to an HI object and increments the reference count for that object.

`Release`
: Decrements the reference count for an HI object and, if the count reaches 0, disposes of any dynamically allocated resources, such as memory, associated with the object.

`GetOwnerCount`
: Returns the current reference count for an HI object.

`Terminate`
: Disposes of any dynamically allocated resources, such as memory, associated with an HI object.

Getting Object Attributes

`GetWindow`
: Returns a pointer to the window object in which an HI object is located.

`GetPort`
: Returns a pointer to the `CGrafPort` record for an object's color graphics port.

`GetRefLabel`
: Gets an HI object's reference label.

`GetCollection` Returns a reference to the collection associated with an HI object.

Getting and Setting an Object's State Change Callback Function

`AddStateChangeCallback`
Adds a given procedure pointer to an HI object's list of state change callbacks.

`RemoveStateChangeCallback`
Removes a specified state change callback function.

Manipulating an HI Object's Size and Location

`GetBoundingRect` Gets an HI object's bounding rectangle—that is, the rectangle that defines its location.

`SetBoundingRect` Sets an HI object's bounding rectangle—that is, the rectangle that defines its location.

`SetPosition` Sets an HI object's position.

`SetSize` Sets an HI object's size.

`CalculateOptimalSize`
Gets an HI object's optimal size.

`GetUpdateRect` Gets an HI object's update rectangle—that is, the rectangle that encloses the entire area in which the object can draw.

Enabling and Disabling an HI Object

`Enable` Enables an HI object.

`Disable` Disables an HI object.

`IsEnabled` Returns an HI object's enabled state.

Getting and Setting an HI Object's Visibility

`Show` Makes an HI object visible.

`Hide` Hides an HI object.

`IsVisible` Returns an HI object's visibility state.

Getting and Setting an HI Object Title's Image Reference

`GetTitle` Gets the image reference for an HI object's title.

`SetTitle` Sets an HI object's title.

Event Handling

`HandleAppleEvent` Handles a given Apple event.

`HandleActivate` Handles a Window Activated event.

`HandleDeactivate` Handles a Window Deactivated event.

`HandleNavigation` Handles a Navigation event.

`HandleMouseDownInContent`
Handles a Mouse Down event in an HI object's bounding rectangle.

`HandleMouseMovedInContent`
Handles a Mouse Moved event in an HI object's bounding rectangle.

`HandleMouseStoppedMovingInContent`
Handles a Mouse Stopped Moving event in an HI object's bounding rectangle.

`HandleMouseUpInContent`
Handles a Mouse Up event in an HI object's bounding rectangle.

Controlling User Input Focus

`TakeUserInputFocus` Assigns user input focus to an HI object.

`ReleaseUserInputFocus`
Releases user input focus.

`HasUserInputFocus` Returns an HI object's user input focus state.

`CanReleaseUserInputFocus`
Indicates whether an HI object can release input focus.

`SetUserInputFocusFlags`
Sets an HI object's user input focus support flags.

`GetUserInputFocusFlags`
Returns an HI object's user input focus support flags.

Imaging

`Draw` Causes an HI object to draw itself in a specified color graphics port.

`Erase` Causes an HI object to erase itself in a specified color graphics port.

`Invalidate` Forces an HI object to redraw itself according to its drawing mode.

`GetDrawingMode`	Gets an HI object's drawing mode.
`SetDrawingMode`	Sets an HI object's drawing mode.
`GetBackgroundPattern`	Returns an HI object's background pattern.
`SetBackgroundPattern`	Sets an HI object's background pattern.

Supporting Clipboard Operations

`Cut`	Cuts current selection.
`Copy`	Copies current selection.
`Paste`	Pastes at the location of the current selection.
`Clear`	Clears the current selection.
`GetClipboardSupportFlags`	Returns an HI object's clipboard support flags.

Summary of Protected Methods

For detailed descriptions of these methods, see "Protected Methods" (page 4-115).

`DrawContent`	Draws an HI object within its update rectangle.
`EraseContent`	Erases an HI object within its update rectangle.
`TranslatePoint`	Translates a given point between coordinate systems.
`TranslateRect`	Translates a given rectangle between coordinate systems.
`StateChanged`	Causes an HI object to invoke its state change functions.
`SetClipboardSupportFlags`	Sets an HI object's clipboard support flags.
`Verify`	Verifies that an HI object's internal state is valid.

Execution Environments

Subclasses of `HIObject` can be instantiated only by the main task of a cooperative program. Methods defined by the HI Objects class library

- are not preemptively reentrant
- cannot be called at secondary interrupt level
- cannot be called at hardware interrupt level

Constants and Data Types

Reference Labels

Reference labels allow your application to assign nonlocalizable names to the HI objects it creates. This makes it possible, for example, to refer to an HI object in a script by the same name no matter how your application is localized.

Reference labels are not used by HI objects themselves. Your application is entirely responsible for assigning and keeping track of its own reference labels. You can use them in any way you want. For example, you can assign one label for each object, for a group of objects, for all your application's HI objects, or whatever other scheme is convenient. You also determine the rules for and enforcement of the uniqueness of reference labels within your application.

When you initialize a new HI object with the `Init` method (page 4-26), you pass the object's reference label as a parameter. When you initialize a new HI object with the `InitFromAERecord` method (page 4-28) or the static method `GetNewHIObject`, the AE record used for initialization specifies the object's reference label.

```
typedef struct RefLabel RefLabel;

struct RefLabel {
    OSType  creator;    /* creator of the object; use application signature */
    OSType  id;         /* label used to identify the object */
};
```

Field descriptions

`creator`	Identifies the creator of the HI object. Use your application's signature in this field to identify the objects it creates.
`id`	Identifies the HI object. You can use any four-character code for the data in this field.

Use the static method `GetHIObjectFromRefLabel` (page 4-23) to find all the HI objects that share a particular reference label within your application's process.

Adoption Flags

The `AddSubPanel` method (defined by `HIEmbeddingPanel`) associates a subpanel with an embedding panel. Class `HIEmbeddingPanel` defines adoption flags of type `HIAdoptionFlags` to indicate how the embedding panel controls the subpanel after adopting it.

```
typedef OptionBits HIAdoptionFlags;
```

Information about `HIEmbeddingPanel` and the adoption flags it defines will be provided with later developer releases.

Drawing Modes

Drawing modes allow your application to control when drawing begins. They determine when the object should redraw itself after a call to the `Invalidate` method (page 4-102): immediately, after the next Update event, or at some later time. Controlling the drawing mode can be useful, for example, if you are adding a number of panels to a dialog box or altering existing panels and you want to make sure drawing occurs only after you're finished.

Drawing modes are identified by enumerators of type `HIDrawingMode`.

```
typedef OptionBits HIDrawingMode;
enum {
    kHIDrawNextUpdateEvent =0x00000000, /* redraw on next Update event; default mode */
    kHIDrawImmediately   =  0x00000001, /* redraw now */
    kHIDeferDrawing      =  0x00000002  /* redraw when mode changes */
};
```

Enumerator descriptions

`kHIDrawNextUpdateEvent`
: Calling `Invalidate` causes the object to redraw itself when it next receives an Update event.

`kHIDrawImmediately`
: Calling `Invalidate` causes the object to redraw itself immediately.

`kHIDeferDrawing`
: Calling `Invalidate` causes the object to redraw itself some time later when the mode is changed to one of the first two values.

To set drawing modes, use the `SetDrawingMode` method (page 4-105).

Coordinate System Constants

The methods `GetBoundingRect` (page 4-48), `SetBoundingRect` (page 4-49), `SetPosition` (page 4-51), and `GetUpdateRect` (page 4-56) allow you to specify the coordinate system used to set or return values related to an HI object's position. You do so with enumerators of type `HICoordinateSystem`.

```
typedef OSType HICoordinateSystem;
enum {
    kHICoordScreenRelative  = 'scrn',
    kHICoordPortRelative    = 'port',
    kHIObjectRelative       = 'obj '
};
```

Enumerator descriptions

`kHICoordScreenRelative`
: Coordinate system relative to the screen. You typically use this coordinate system with `HIWindow` objects only.

`kHICoordPortRelative`
: Coordinate system relative to the graphics port.

`kHICoordObjectRelative`
: Coordinate system relative to the object itself.

Subclasses of HIObject may declare other coordinate systems that apply to those classes. If you are implementing a subclass that declares its own coordinate system, you can use the protected methods `TranslatePoint` (page 4-118) or `TranslateRect` (page 4-120) to translate between coordinate systems.

User Input Focus Support Flags

The `GetUserInputFocusFlags` (page 4-97) and `SetUserInputFocusFlags` (page 4-96) methods allow you to set flags of type `HIUserInputFocusFlags` that determine an object's ability to accept user input focus.

```
typedef OptionBits HIUserInputFocusFlags;
enum {
    kHICanTakeUserInputFocus            =   0x00000001,
    kHITakesUserInputFocusWhenClicked   =   0x00000002
};
```

Constant descriptions

`kHICanTakeUserInputFocus`
: The object can accept user input focus. This flag typically doesn't change over the life of the object. Any interactive object that reacts to text entry and navigation events should set this flag.

`kHITakesUserInputFocusWhenClicked`
: The object accepts user input focus when clicked. Some objects (for example, editable text fields) should acquire focus when the user clicks their content. Other objects (for example, push buttons and checkboxes) should not accept focus when clicked.

Clipboard Support Flags

Some kinds of HI objects allow a user to perform Cut, Copy, Paste, Clear, or Undo operations while the object has user input focus. For example, an editable text panel can support any of these operations at different times, depending on the user's most recent action.

To determine whether an HI object currently has user focus, your application calls the `HasUserInputFocus` method. If the object has user input focus but no text is selected, the application should disable the Cut, Copy, and Clear commands in the Edit menu and may or may not disable the Paste and Undo commands, depending on whether the Clipboard currently contains a scrap and on the user's most recent action. If the object has user input focus and some text is selected, the application should ensure that the Cut, Copy, and Clear commands are enabled.

The public method `GetClipboardSupportFlags` (page 4-114) returns flags of type `HIClipboardSupportFlags` that provide information about an object's current state with respect to the Clipboard.

```
typedef OptionBits HIClipboardSupportFlags;
enum {
    kHISupportsCut      = 0x00000001, /* object supports cutting */
    kHISupportsCopy     = 0x00000002, /* object supports copying */
    kHISupportsPaste    = 0x00000003, /* object supports pasting */
    kHISupportsClear    = 0x00000004  /* object supports clearing */
};
```

Each HI object is responsible for keeping track of its own state with respect to Clipboard-related commands and for using the protected method `SetClipboardSupportFlags` to set the flags that reflect that state.

State Change Callback Function

You can use the `AddStateChangeCallback` method (page 4-44) to add a state change callback function to an HI object's collection of such functions. When an object's state changes—for example, when a user selects a checkbox—the object calls each of its associated state change callback functions to perform whatever action is appropriate. This function allows you to associate actions with a specific HI object state without having to create a subclass.

```
typedef void (*HIStateChangeCallbackProcPtr)(
                    Environment *ev,
                    HIStateChangeCodeCreator selectorCreator,
                    HIStateChangeCode whatHappened,
                    HIObject *theObject);
```

The HI object passes a state change code creator and a state change code in the `selectorCreator` and `whatHappened` parameters, respectively. For information about these types and their enumerations, see "State Change Codes" (page 4-16).

When you use the `AddStateChangeCallback` method (page 4-44) to associate a changed state function with an HI object, it creates a state change callback reference that refers to the function.

```
typedef struct OpaqueHIStateChangeCallbackRef* HIStateChangeCallbackRef;
```

You pass the callback reference `HIChangedStateCallbackRef` to `RemoveChangeStateCallback` (page 4-46) to remove a particular callback.

For information about writing your own state change callback function, see "Application-Defined Function" (page 4-125).

State Change Codes

You use the `AddStateChangeCallback` method (page 4-44) to associate a state change callback function with an HI object. A given object may have multiple state change callback functions installed by different clients.Whenever an object's state changes, the object calls all its state change callback functions, passing them information that identifies the change in state that has occurred. Each kind of state change is uniquely identified by two IDs: a state change code creator and a state change code.

The code creator identifies the name space for state changes defined by a particular creator. The state change identifies a unique state change defined by that creator. You should use your application signature as the creator code for any state change codes defined by your application.

Class `HIObject` defines a creator code for the state changes defined by Apple Computer and several state change codes.

```
typedef OSType HIStateChangeCodeCreator; /* creator ID */
enum {
    kHIObjectAppleCreator           =   'aapl'
}

typedef OSType HIStateChangeCode;   /* code indicating what changed */
enum {
    kHIStateChangeObjectBoundsChanging      =   'bnd1',
    kHIStateChangeObjectBoundsChanged       =   'bnd2',
    kHIStateChangeObjectVisibilityChanging  =   'vis1',
    kHIStateChangeObjectVisibilityChanged   =   'vis2'
};
```

Enumerator descriptions

`kHIObjectAppleCreator`

The state change code creator for the state codes defined by Apple.

`kHIStateChangeObjectBoundsChanging`
The object's bounds are about to change.

`kHIStateChangeObjectBoundsChanged`The object's bounds have changed.

`kHIStateChangeObjectVisibilityChanging`
The object's visibility is about to change.

`kHIStateChangeObjectVisibilityChanged`
The object's visibility has changed.

Only HI object classes define creator codes. The HI object specifies the state change creator and the appropriate state change code when it calls the protected method `StateChanged` (page 4-122). A callback function can react to state change codes defined by any creator.

For information about state change callback functions, see "State Change Callback Function" (page 4-15) and "Application-Defined Function" (page 4-125).

AE Record Keywords

The HI Objects class library uses the Apple Event Manager data structure called an AE record as the universal format for describing a HI objects. This hierarchical data format permits the precise description of the initialization data for all the subclasses in an HI object's inheritance hierarchy.

An AE record that describes an HI object typically specifies the object's class, the data format used in the AE record, and all the data required to initialize the object, including its reference label, visibility state, background pattern, title, and so on. You can initialize a new HI object using such an AE record; convert it to a flattened stream of data suitable for storage in a resource; send it as part of an Apple event; or keep it in memory as a way of maintaining information about an object without keeping the object itself in memory.

The HI object initialization data in an AE record consists of a series of nested AE records, each describing the initialization data for that class and specifying another AE record that describes the initialization data for its subclass. For example, the nested AE records describing a push button consist of an AE record for data defined by `HIObject`, which contains an AE record for data defined by `HIPanel`, which contains an AE record for data defined by `HIControl`, which contains an AE record for data defined by `HIPushButton`.

Information about the keywords used in AE records that describe HI object classes will be available with later developer releases.You typically use a

resource editor to create resources that contain flattened AE records describing HI objects.

For information about `HIObject` methods that use AE records, see "Static Methods" (page 4-19) and "Initializing, Saving, and Disposing of an Object" (page 4-25).

AE Record Data Formats

The methods `InitFromAERecord` (page 4-28) and `WriteToAERecord` (page 4-31) include a parameter of type `HIAERecordDataFormat` that specifies the format to use for reading or writing the AE record.

```
typedef OptionBits HIAERecordDataFormat;
enum {
    kHIUseResourceReferences    = 0x00000000,
    kHIFlattenAllData           = 0x00000001,
    kHISimpleValuesOnly         = 0x00000002
};
```

Constant descriptions

`kHIUseResourceReferences`
: The data consists of reference to resources that contain the object data. For example, if the human interface for an HI object includes text (such as a title), the text should be stored in a text object resource, and the resource ID should be stored in the AE record.

`kHIFlattenAllData`
: The data consists of a flattened representation of the object data itself, which is encapsulated in an AE record. For example, if the human interface for an HI object includes text, the text object for the text should be flattened into a buffer and stored directly in the AE record.

`kHISimpleValuesOnly`
: The data consists of simple values. It provides only the information required to express the object's intrinsic value. For example, the AE record for an `HIControl` object includes its 32-bit control value but not its bounding rectangle or title. You can't use this data format when you first initialize an HI object, but you can use it if you are reinitializing the object.

Static Methods

The HI Objects class library uses AE records to describe HI objects. An AE record that describes an object typically specifies the object's class, the data format used in the AE record, and all the data required to initialize the object. For more information about the use of AE records to describe HI objects, see "AE Record Keywords" (page 4-17).

The static method `GetNewHIObject` (page 4-21) instantiates an HI object based on information in an AE record, then passes a reference to the AE record to the object's `InitFromAERecord` method (page 4-28). `InitFromAERecord` uses the initialization data in the AE record to initialize the object.

To instantiate and initialize an HI object from data stored in a resource, use the static method `GetNewHIObjectFromResource` (page 4-19). `GetNewHIObjectFromResource` reads a resource into memory, converts it to an AE record, and passes a reference to the AE record to `GetNewHIObject`.

To retrieve all the HI objects in the current process that have a specified reference label, use the static method `GetHIObjectFromRefLabel` (page 4-23).

For more information about initializing HI objects, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

GetNewHIObjectFromResource

Reads a resource into memory and uses its data to instantiate and initialize a new HI object.

IDL DECLARATION

```
GetNewHIObjectFromResource (
                    in ResID objectInitializerResourceID,
                    out HIObject newObject);
```

C DECLARATION

```
OSStatus HIObject_GetNewHIObjectFromResource (
                    Environment *ev,
                    ResID objectInitializerResourceID,
                    HIObject **newObject);
```

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`objectInitializerResourceID`
The resource ID for the resource from which to get the description of the new HI object.

`newObject` A pointer to an HI object pointer. On output, the HI object pointer identifies the new initialized object.

function result Information about result codes will be provided with later developer releases.

DESCRIPTION

`GetNewHIObjectFromResource` reads a specified resource into a buffer. The only purpose of the resource is to supply data that `GetNewHIObjectFromResource` converts from the flattened format used in resources to the AE record format. `GetNewHIObjectFromResource` manages all aspects of its use of the resource, including disposal.

To allocate and initialize the HI object described by the AE record, `GetNewHIObjectFromResource` calls `GetNewHIObject` (page 4-21).

▲ **WARNING**
The new HI object identified by the `newObject` parameter is a SOM object, not a resource. Don't attempt to use Resource Manager calls on it. ▲

SPECIAL CONSIDERATIONS

Every call to `GetNewHIObjectFromResource` must be matched by an equivalent call to `Release` (page 4-35).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

SEE ALSO

For an overview of related static methods, see "Static Methods" (page 4-19).

For an overview of related public methods and the object life cycle management provided by `HIObject`, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

GetNewHIObject

Instantiates and initializes an HI object using data in an AE record.

IDL DECLARATION

```
OSStatus GetNewHIObject (
                    in AESubDesc objectInitializer,
                    out HIObject newObject);
```

C DECLARATION

```
OSStatus HIObject_GetNewHIObject (
                    Environment *ev,
                    AESubDesc *objectInitializer,
                    HIObject **newObject);
```

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller.

`objectInitializer`
A pointer to an Apple event subdescriptor. On input, the subdescriptor identifies an AE record that describes an HI object. For more information, see "AE Record Keywords" (page 4-17).

`newObject` A pointer to an HI object pointer. On output, the HI object pointer identifies the new initialized object of the requested class.

function result Information about result codes will be provided with later developer releases.

DESCRIPTION

`GetNewHIObject` determines what class of object to create and what kind of data format to use based on information in the AE record identified by the `objectInitializer` parameter, then uses SOM techniques to create that object. Next, `GetNewHIObject` uses the Apple Event Manager to create a subdescriptor identifying the portion of the AE record that contains the actual initialization data and passes that subdescriptor to the `InitFromAERecord` method (page 4-28) to initialize the object. You can also use `InitFromAERecord` directly if the corresponding instantiated object already exists.

If you are creating an HI object from data in a resource, you don't need to use `GetNewHIObject`. Instead, use `GetNewHIObjectFromResource` (page 4-19).

SPECIAL CONSIDERATIONS

Every call to `GetNewHIObject` must be matched by an equivalent call to `Release` (page 4-35).

CALLING RESTRICTIONS

Static methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

SEE ALSO

For an overview of related static methods, see "Static Methods" (page 4-19).

For an overview of related public methods and the object life cycle management provided by `HIObject`, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

GetHIObjectFromRefLabel

Finds all the HI objects in the current process that have a specified reference label.

IDL DECLARATION

```
OSStatus GetHIObjectFromRefLabel (
                    in RefLabel identifier,
                    in ItemCount requestedObjects,
                    out ItemCount totalObjects,
                    inout HIObject theObjects);
```

C DECLARATION

```
OSStatus HIObject_GetHIObjectFromRefLabel(
                    Environment *ev,
                    RefLabel *identifier,
                    ItemCount requestedObjects,
                    ItemCount *totalObjects,
                    HIObject **theObjects);
```

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`identifier` A pointer to a reference label. On input, you specify a reference label for the objects you want to retrieve. See "Reference Labels" (page 4-11) for details.

`requestedObjects`
The number of HI objects the array specified by the `theObjects` parameter can hold. `GetHIObjectFromRefLabel` returns no more than this number of objects.

`totalObjects`	A pointer to an item count. On output, the count reflects the total number of HI objects with the specified reference label that actually exist. If you don't want the item count returned, set `totalObjects` to `NULL`.
`theObjects`	A pointer to an array of HI object pointers. On output, this array identifies the requested objects that it has room for (up to the number that actually exist). `GetHIObjectFromRefLabel` assumes that the array has at least enough space to hold pointers for the number of requested HI objects indicated by the `requestedObjects` parameter. If you don't want the HI objects returned, set `theObjects` and `requestedObjects` to `NULL`.
function result	Information about result codes will be provided with later developer releases.

DISCUSSION

When you initialize a new HI object, you can assign it a reference label of any value. This can be useful for keeping track of related HI objects, such as identical controls that appear in several dialog boxes.

You can also assign a unique reference label to an HI object so you can locate it easily with `GetHIObjectFromRefLabel`. For example, if you need to disable the OK button in a modeless dialog box, you can locate it by calling `GetHIObjectFromRefLabel`, then disable it by calling the button's `Disable` method.

To obtain the total number of HI objects with a specified reference label before allocating memory for `theObjects`, set `requestedObjects` and `theObjects` to `NULL`.

CALLING RESTRICTIONS

Static methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

SEE ALSO

For an overview of related static methods, see "Static Methods" (page 4-19).

Public Methods

Initializing, Saving, and Disposing of an Object

The HI Objects class library uses AE records to describe HI objects. An AE record that describes an object typically specifies the object's class, the data format used in the AE record, and all the data required to initialize the object. For more information about AE records and their uses, see "AE Record Keywords" (page 4-17).

The static method `GetNewHIObject` (page 4-21) instantiates an HI object based on information in an AE record, then passes a reference to the AE record to the object's `InitFromAERecord` method (page 4-28). `InitFromAERecord` uses the initialization data in the AE record to initialize the object. If an AE record for an HI object and the corresponding instantiated object already exist, you can pass a reference to it to `InitFromAERecord` directly.

You can use the `WriteToAERecord` method (page 4-31) to create a persistent representation of any HI object as an AE record (the inverse of `InitFromAERecord`). This can be useful if you want to keep an AE record for an object in memory so you can reinitialize the object at a later time. Resource editors use `WriteToAERecord` to create AE records for storage in resources.

To instantiate and initialize an HI object from data stored in a resource, you use the static method `GetNewHIObjectFromResource` (page 4-19). `GetNewHIObjectFromResource` reads a resource into memory, converts it to an AE record, and passes a reference to the AE record to `GetNewHIObject`.

To initialize an object programmatically, you use the initialization method provided by that object's class—for example, `InitDialog` for dialog panels or `InitRadioButtonGroup` for radio button group panels. The implementation of each such method calls the initialization method of its superclass, which calls the initialization method of its superclass, and so on until `Init` (page 4-26) gets called. `Init` is normally not called directly by a client of the HI Objects class library. After initializing the object, you can use its methods to set characteristics such as its visibility and title, handle events, or perform any other actions the object knows how to perform.

It is often desirable to use the same HI object for a variety of purposes within an application. The HI Objects class library provides object life cycle management for all HI objects. To do so, it keeps track of references to every HI object your application creates, incrementing the object's reference count

whenever a new reference gets created and decrementing the count whenever a reference gets released.

To get a new reference to an existing HI object, use the `Clone` method (page 4-33). `Clone` increments the reference count for the HI object and returns a new pointer to the object. When you are finished with any reference to an HI object, call the `Release` method. `Release` decrements the reference count, calling the `Terminate` method (page 4-37) to dispose of the original object only when the reference count reaches 0.

Every call to `GetNewHIObject`, `GetNewHIObjectFromResource`, or `Clone` must be matched by an equivalent call to `Release`.

You shouldn't normally call `Terminate` directly. Instead, you should call `Release` to take advantage of its support for reference counting. Call `Terminate` directly only if you want to clear out an object's state before reinitializing it with a call to `InitFromAERecord`.

Init

Initializes an existing HI object programmatically.

IDL DECLARATION

```
OSStatus Init       (in RefLabel identifier,
                     in Rect bounds);
```

C DECLARATION

```
OSStatus HIObject_Init (
                     HIObject *somSelf,
                     Environment *ev,
                     RefLabel *identifier
                     Rect *bounds);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
`identifier`	A pointer to a reference label. On input, you specify a reference label. See "Reference Labels" (page 4-11) for details.
`bounds`	A pointer to a rectangle. On input, you specify the rectangle in which you want to create the object. The default coordinate system for this rectangle is dictated by the subclass. For example, for windows this rectangle is in screen-relative coordinates, while for panels it is in port-relative coordinates.
function result	Information about result codes will be provided with later developer releases.

DISCUSSION

To initialize an object programmatically, you use the initialization method provided by that object's class—for example, `InitDialog` for dialog panels or `InitRadioButtonGroup` for radio button group panels. The implementation of each such method calls the initialization method of its superclass, which calls the initialization method of its superclass, and so on until `Init` gets called. `Init` is normally not called directly by a client of the HI Objects class library.

You must initialize an HI object successfully before you can call any other method on that object. In addition to initializing an existing HI object programmatically, you can create and initialize an HI object using `GetNewHIObjectFromResource` (page 4-19) or `GetNewHIObject` (page 4-21), and you can initialize an existing HI object using `InitFromAERecord` (page 4-28).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

`Init` should be called by subclasses of class `HIObject` only.

OVERRIDE INFORMATION

If you subclass from class `HIObject`, do not override `Init`. Instead, add a new initialization method for your subclass. The programmatic initialization

method for any subclass of `HIObject` must first call the programmatic initialization method defined by its immediate superclass, and if that call is successful, complete any initialization work required to make the object usable. Note that any resources allocated inside the initialization method must be deallocated inside the `Terminate` method (page 4-37).

SEE ALSO

For an overview of related methods, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

InitFromAERecord

Initializes an HI object on the basis of data in an AE record.

IDL DECLARATION

```
OSStatus InitFromAERecord (
                    in AESubDesc baseInitializerData,
                    in AESubDesc initializerDataForClass,
                    in HIObjectDataFormat dataFormat);
```

C DECLARATION

```
OSStatus HIObject_InitFromAERecord (
                    HIObject *somSelf,
                    Environment *ev,
                    AESubDesc *baseInitializerData,
                    AESubDesc *initializerDataForClass,
                    HIAERecordDataFormat dataFormat);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`baseInitializerData`
A pointer to an Apple event subdescriptor. On input, you provide a subdescriptor that identifies the AE record for the entire HI object. For details, see "AE Record Keywords" (page 4-17).

`initializerDataForClass`
A pointer to an Apple event subdescriptor. On input, class implementations provide a subdescriptor that points to initializer data for the next subclass in the object's inheritance hierarchy. Every call to `HIAERecordFormat` must provide a subdescriptor of some kind in this parameter.

`dataFormat` The data format used in the AE record specified by the `baseInitializerData` parameter. You identify the data format with one of the values defined in the `HIAERecordDataFormat` enumeration (page 4-18)—with the exception of `kHISimpleValuesOnly`, which can be used only with the `WriteTOAERecord` method. (For more details, see the discussion that follows.)

function result Information about result codes will be provided with later developer releases.

DISCUSSION

You don't usually call `InitFromAERecord` directly. Instead, you pass an AE record describing the object to the static method `GetNewHIObject` (page 4-21), which instantiates the new HI object and passes a reference to the AE record to the object's `InitFromAERecord` method. `InitFromAERecord` uses the initialization data in the AE record to initialize the object. If an AE record for an HI object and the corresponding instantiated object already exist, you can pass a reference to it directly to `InitFromAERecord`.

When your application calls `InitFromAERecord`, the method ignores the contents of the `initializerDataForClass` parameter. On output, the implementation of `InitFromAERecord` for class `HIObject` sets up the `initializerDataForClass` parameter to point to initializer data for the next subclass in the object's inheritance hierarchy. The implementation of `InitFromAERecord` for the subclass

then uses that data to perform its own initialization tasks. For more details, see the override information that follows.

If you set the `dataFormat` parameter to `kHIUseResourceReferences`, the AE record identified by the subdescriptor record must contain resource IDs for data that is typically encapsulated in resources (such as text objects, pictures, pixel patterns, and so on). If `dataFormat` is set to `kHIFlattenAllData`, the AE record must contain all of the actual data, and `AEInitFromAERecord` won't look for resource IDs.

Because it simplifies localization, you typically use the `kHIUseResourceReferences` data format for storing and instantiating HI objects inside your application. You use the `kHIFlattenAllData` data format only if your application cannot assume that its resource fork is present when the object is instantiated.

You cannot use the `kHISimpleValuesOnly` data format when you first initialize an object. However, you can reset the values of an object that has already been initialized by calling `InitFromAERecord` again on the same object and specifying `kHISimpleValuesOnly`. This can be useful if you want to reuse an existing object with different initial values.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

To support initialization from an AE record, each subclass of `HIObject` must override `InitFromAERecord`. The overridden method always calls its inherited `InitFromAERecord` method before performing its own initialization tasks. If the inherited call executes successfully (that is, returns `noErr`), you can assume that the subdescriptor in the `initializerDataForClass` parameter points to the AE record containing the initialization data for your subclass implementation.

After performing its own initialization tasks, the `InitFromAERecord` method for any subclass must set up the `initializerDataForClass` parameter so that it points to the initializer data for the next subclass in the object's inheritance hierarchy, if any. To do so, each subclass typically uses code like this:

```
myError = AEGetKeySubDesc (dataForCurrentClass,
                           keyHISubclassInitializer,
                           dataForCurrentClass);
```

If the subclass is at the bottom of the object's inheritance hierarchy (that is, it is the object's own class), `AEGetKeySubDesc` returns `errAEDescNotFound`. If the `HIObject` (or any abstract class) implementation of `InitFromAERecord` receives this result code, it also returns `errAEDescNotFound`, indicating that something is wrong with the data or that an attempt has been made to initialize an object from an abstract class, and subclass implementations should also return `errAEDescNotFound`. If a concrete subclass implementation of `InitFromAERecord` receives this result code, it should return `noErr`.

SEE ALSO

For an overview of related methods, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

To instantiate and initialize an HI object from data stored in a resource, use the static method `GetNewHIObjectFromResource` (page 4-19).

WriteToAERecord

Writes an HI object's state and initialization data to an AE record that can be saved on disk, sent as part of an Apple event, or passed to `InitFromAERecord`.

IDL DECLARATION

```
OSStatus WriteToAERecord (in AERecord dataForSubclass,
                          inout AERecord objectData,
                          in HIObjectDataFormat dataFormat);
```

C DECLARATION

```
OSStatus HIObject_WriteToAERecord (
                          HIObject *somSelf,
                          Environment *ev,
```

```
                    AERecord *dataForSubclass,
                    AERecord *objectData,
                    HIAERecordDataFormat dataFormat);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`dataForSubclass`
A pointer to an AE record. On input, `WriteToAERecord` assumes that this AE record contains all the data for the object's subclasses in the format specified by the `dataFormat` parameter. If the object has no subclasses, set this parameter to `NULL`.

`objectData` A pointer to an AE record. On output, this record contains all the data required to initialize the object in the state it's in when you call `WriteToAERecord`. For details, see "AE Record Keywords" (page 4-17).

`dataFormat` The data format used for AE records specified by the `dataForSubclass` and `objectData` parameters. You identify the data format with one of the values defined in the `HIAERecordDataFormat` enumeration (page 4-18).

function result Information about result codes will be provided with later developer releases.

DISCUSSION

`WriteToAERecord` creates a persistent representation of any HI object as an AE record (the inverse of `InitFromAERecord`). This can be useful if you want to keep an AE record for an object in memory so you can instantiate and reinitialize the object at a later time. For example, a dialog box that can display a series of pages can save the state of each page in an AE record, then call `GetNewHIObject` to instantiate and initialize the page when the user chooses to display it.

Resource editors use `WriteToAERecord` to create AE records for storage in resources.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

A subclass of `HIObject` must override `WriteToAERecord` to support initialization from data in an AE record—the standard way of instantiating HI objects. The implementation of the override must perform the following steps:

1. Create an AE record, `dataForClass`, and attach all class data to that AE record in the format specified in the `dataFormat` parameter.
2. If `dataForSubclass` is not null, copy that data as a subrecord to `dataForClass`. Dispose of `dataForSubclass`.
3. Call the inherited `WriteToAERecord` method. Pass `dataForClass` as the `dataForSubclass` parameter.

SEE ALSO

For an overview of related methods, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

Clone

Returns a new pointer to an HI object and increments the reference count for that object.

IDL DECLARATION

```
HIObject Clone ();
```

C DECLARATION

```
HIObject *HIObject_Clone (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

function result A new pointer to the object.

DISCUSSION

You use `Clone` to create a duplicate reference to an HI object. Every call to `Clone` must be matched by a call to `Release`.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `Clone`.

SEE ALSO

For an overview of related methods, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

Release

Decrements the reference count for an HI object and, if the count reaches 0, disposes of any dynamically allocated resources, such as memory, associated with the object.

IDL DECLARATION

```
void Release ();
```

C DECLARATION

```
void HIObject_Release (HIObject *somSelf,
                       Environment *ev);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

When you are finished with any reference to an HI object, you should call `Release` to decrement its reference count. When a call to `Release` brings an HI object's reference count to 0, `Release` calls the protected method `Terminate` to dispose of any dynamically allocated resources, such as memory, associated with the object, then uses SOM techniques to dispose of the object itself.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `Release`.

SEE ALSO

For an overview of related methods, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

GetOwnerCount

Returns the current reference count for an HI object.

IDL DECLARATION

```
ItemCount GetOwnerCount ();
```

C DECLARATION

```
ItemCount *HIObject_GetOwnerCount (HIObject *somSelf,
                    Environment *ev);
```

`somSelf`	A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
function result	A pointer to an item count that reflects the current reference count for the object.

DISCUSSION

`GetOwnerCount` is used only for debugging and for certain kinds of exception handling.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetOwnerCount`.

SEE ALSO

For an overview of related methods, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

Terminate

Disposes of any dynamically allocated resources, such as memory, associated with an HI object.

IDL DECLARATION

```
void Terminate ();
```

C DECLARATION

```
void HIObject_Terminate (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

To release an HI object, you should normally call `Release`, which calls `Terminate` only when the reference count reaches 0.

You should call `Terminate` directly only if you want to clear out an object's state before reinitializing it with a call to `InitFromAERecord`.

After you call Terminate, no other HI object methods can be counted on to work correctly except for those declared in the `SOMObject` base class and the standard HI object initialization methods.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

All HI objects that dynamically allocate memory must override `Terminate` to ensure that the memory is properly freed. The implementation of `Terminate` must perform all of its own termination duties first and then call its inherited `Terminate`.

All HI objects must be robust enough to handle multiple terminations. Typically, this involves setting any pointers to `NULL` after disposing of them and checking for `NULL` pointers before disposing of them.

SEE ALSO

For an overview of related methods, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

Getting HI Object Attributes

GetWindow

Returns a pointer to the window object in which an HI object is located.

IDL DECLARATION

```
HIWindow GetWindow ();
```

C DECLARATION

```
HIWindow* HIObject_GetWindow (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf`	A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
function result	A pointer to the window object in which the HI object is located.

DISCUSSION

All HI objects are associated with windows. If the HI object itself is a window, then its window is itself. Otherwise, the window is the window in which the object typically displays itself and interacts with the user.

Note that while all HI objects are bound to windows, they can be drawn in any color graphics port (for example, when printing or when drawing in an offscreen graphics world).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

It's not usually necessary to override `GetWindow`.

SEE ALSO

Compare `GetPort` (page 4-40).

GetPort

Returns a pointer to the `CGrafPort` record for an HI object's color graphics port.

IDL DECLARATION

```
CGrafPtr GetPort();
```

C DECLARATION

```
CGrafPtr HIObject_GetPort (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

function result A pointer to the `CGrafPort` record for the object's graphics port.

DISCUSSION

`GetPort` simply identifies the color graphics port associated with the HI object's window.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetPort`.

SEE ALSO

Compare `GetWindow` (page 4-39).

GetRefLabel

Gets an HI object's reference label.

IDL DECLARATION

```
void GetRefLabel (out RefLabel identifier);
```

C DECLARATION

```
void HIObject_GetRefLabel (
                    HIObject *somSelf,
                    Environment *ev,
                    RefLabel *identifier);
```

`somSelf` A pointer to the HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
`identifier`	A pointer to a reference label. On output, this parameter identifies the object's reference label. See "Reference Labels" (page 4-11) for details.

DISCUSSION

A reference label allows your application to identify any HI object by means of a static nonlocalizable value of your choice. You assign a reference label to an object when you initialize it.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetRefLabel`.

GetCollection

Returns a reference to the collection associated with an HI object.

IDL DECLARATION

```
Collection GetCollection();
```

C DECLARATION

```
Collection HIObject_GetCollection (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf`	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
function result	A reference to the collection associated with the object. You can use Collection Manager functions to manipulate the collection.

DISCUSSION

`GetCollection` returns a collection reference that you can pass to Collection Manager functions to get, add, or remove collection items associated with the object. Collection items can be useful for adding application-specific data to an HI object without having to subclass.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetCollection`.

SEE ALSO

Documentation for the related `HIPanel` method `GetDeepCollectionItemData` will be available with later developer releases. `GetDeepCollectionItemData` searches the HI object and, if the search isn't successful, continues to search the object's embedding parents for a specified collection item.

Getting and Setting an HI Object's State Change Callback Function

AddStateChangeCallback

Adds a given procedure pointer to an HI object's list of state change callbacks.

IDL DECLARATION

```
OSStatus AddStateChangeCallback (
      in HIStateChangeCallbackProcPtr *changedStateCallback,
      out HIStateChangeCallbackRef newCallbackRef);
```

C DECLARATION

```
OSStatus HIObject_AddStateChangeCallback (
                    HIObject *somSelf,
                    Environment *ev,
                    HIStateChangeCallbackProcPtr changedStateCallback,
                    HIStateChangeCallbackRef *newCallbackRef);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`changedStateCallback`
A pointer to the state change callback function you want to add.

`newCallbackRef`
A pointer to a state change callback reference. On output, this reference identifies the state change callback function you have associated with the object. To remove the corresponding callback function pass the reference to

`RemoveStateChangeCallback` (page 4-46). If you set `newCallbackRef` to `NULL`, `AddStateChangeCallback` doesn't return a reference in this parameter.

function result Information about result codes will be provided with later developer releases.

DISCUSSION

`AddStateChangeCallback` adds a state change callback function to the specified object. When the object's state changes—for example, when a user selects a checkbox—the object calls its state change callback functions to perform whatever application-specific action is appropriate.

Many clients can add state change callbacks to an HI object—including the operating system, your application, application plug-ins, and shared libraries. `AddStateChangeCallback` makes no effort to manage the order in which it calls an object's state change callbacks.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `AddStateChangeCallback`.

SEE ALSO

To remove an object's state change callback function, use the `RemoveStateChangeCallback` method (page 4-46).

For information about writing a state change callback function, see "Application-Defined Function" (page 4-125).

For a simple example of the use of a state change callback function, see "Assembling Embedding Panels" (page 1-55).

RemoveStateChangeCallback

Removes a specified state change callback function.

IDL DECLARATION

```
OSStatus RemoveStateChangeCallback (
                    in HIStateChangeCallbackRef callbackRefToRemove);
```

C DECLARATION

```
OSStatus HIObject_RemoveStateChangeCallback (
                    HIObject *somSelf,
                    Environment *ev,
                    HIStateChangeCallbackRef callbackRefToRemove);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`callbackRefToRemove`
A reference to the state change callback function you want to remove.

function result Information about result codes will be provided with later developer releases.

DISCUSSION

The `AddStateChangeCallback` method (page 4-44) assigns a state change callback reference to each callback function that it successfully installs. This reference is necessary because a particular HI object may have more than one state change callback function associated with it. You pass the reference to `RemoveStateChangeCallback` to remove the corresponding callback function.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `RemoveStateChangeCallback`.

SEE ALSO

For information about writing a state change callback function, see "Application-Defined Function" (page 4-125).

Manipulating an HI Object's Size and Location

Class `HIObject` defines several methods that get or set different aspects of an object's size and location:

- To get or set an object's bounding rectangle, use `GetBoundingRect` (page 4-48) or `SetBoundingRect` (page 4-49). You can also change the bounding rectangle by using `SetPosition` (page 4-51) and `SetSize` (page 4-53).
- To get an object's optimal size, use `CalculateOptimalSize` (page 4-54). This is useful with objects whose size may not be easily determined until runtime, such as caption panels.
- To get the rectangle within which an object can draw (which may be larger than its bounding rectangle), use `GetUpdateRect` (page 4-56).

GetBoundingRect

Gets an HI object's bounding rectangle—that is, the rectangle that defines its location.

IDL DECLARATION

```
void GetBoundingRect(
                    in HICoordinateSystem coordinate,
                    out Rect bounds);
```

C DECLARATION

```
void HIObject_GetBoundingRect (
                    HIObject *somSelf,
                    Environment *ev,
                    HICoordinateSystem coordinate,
                    Rect *bounds);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`coordinate` The coordinate system you want `GetBoundingRect` to use to specify the rectangle returned in the `bounds` parameter. You identify the coordinate system with one of the values defined in the `HICoordinateSystem` enumeration (page 4-13).

`bounds` A pointer to a rectangle. On output, the rectangle describes the object's bounds in the coordinate system specified by the `coordinate` parameter.

DISCUSSION

Every HI object has a bounding rectangle that defines its location. The object's content must be drawn within this rectangle; for example, the area of a button

that can respond to mouse clicks corresponds to its bounding rectangle. The object may also draw adornments inside a second, larger rectangle called the update rectangle, which you can obtain by calling the `GetUpdateRect` method (page 4-56).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You can override `GetBoundingRect` if you want to modify or limit the bounding rectangle of an HI object.

SEE ALSO

For an overview of related methods, see "Manipulating an HI Object's Size and Location" (page 4-47).

SetBoundingRect

Sets an HI object's bounding rectangle—that is, the rectangle that defines its location.

IDL DECLARATION

```
void SetBoundingRect (
                    in HICoordinateSystem coordinate,
                    in Rect newBounds);
```

C DECLARATION

```
void HIObject_SetBoundingRect (HIObject *somSelf,
                    Environment *ev,
                    HICoordinateSystem coordinate,
                    Rect* newBounds);
```

somSelf	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
ev	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
coordinate	The coordinate system you want SetBoundingRect to use to interpret the rectangle in the newBounds parameter. You identify the coordinate system with one of the values defined in the HICoordinateSystem enumeration (page 4-13).
newBounds	A pointer to a rectangle. On input, you supply a rectangle describing the bounds to which to set the object in the coordinates specified by the coordinate parameter.

DISCUSSION

Every HI object has a bounding rectangle that defines its location. The object's content must be drawn within this rectangle; for example, the area of a button that can respond to mouse clicks corresponds to its bounding rectangle. The object may also draw adornments inside a second, larger rectangle called the update rectangle, which you can obtain by calling the GetUpdateRect method (page 4-56).

You can change an object's position and size at the same time by calling SetBoundingRect. If you want to change just the object's position or just its size, use the SetPosition (page 4-51) or SetSize (page 4-53) method.

CALLING RESTRICTIONS

Methods defined by HIObject and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `SetBoundingRect`. If you want to modify the behavior of an HI object when its position or size is changed, override `SetPosition` (page 4-51) or `SetSize` (page 4-53).

SEE ALSO

For an overview of related methods, see "Manipulating an HI Object's Size and Location" (page 4-47).

SetPosition

Sets an HI object's position.

IDL DECLARATION

```
void SetPosition    (in HICoordinateSystem coordinate,
                     in Point newPosition);
```

C DECLARATION

```
void HIObject_SetPosition (
                     HIObject *somSelf,
                     Environment *ev,
                     HICoordinateSystem coordinate,
                     Point *newPosition);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`coordinate`	The coordinate system you want `SetPosition` to use to interpret the point in the `newPosition` parameter. You identify the coordinate system with one of the values defined in the `HICoordinateSystem` enumeration (page 4-13).
`newPosition`	A pointer to a point. On input, you supply the point describing the position of the object's upper-left corner in the coordinate system specified by the `coordinate` parameter.

DISCUSSION

Because `SetPosition` changes the location of an HI object, its implementation ensures that the appropriate screen areas are erased and invalidated.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You can override `SetPosition` if you would like your object to modify its behavior when its position changes. Your implementation of `SetPosition` should call the inherited `SetPosition` method to maintain the object's geometry correctly.

SEE ALSO

For an overview of related methods, see "Manipulating an HI Object's Size and Location" (page 4-47).

SetSize

Sets an HI object's size.

IDL DECLARATION

```
void SetSize        (in SInt16 width,
                     in SInt16 height);
```

C DECLARATION

```
void HIObject_SetSize (
                     HIObject *somSelf, Environment *ev,
                     SInt16 width,
                     SInt16 height);
```

somSelf	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
ev	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
width	The width to which you want to set the object, in pixels.
height	The height to which you want to set the object, in pixels.

DISCUSSION

Because `SetSize` changes the location of an HI object, its implementation ensures that the appropriate screen areas are erased and invalidated.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You can override `SetSize` if you would like your object to modify its behavior when its size changes. Your implementation of `SetSize` should call the inherited `SetSize` method to maintain the object's geometry correctly.

SEE ALSO

For an overview of related methods, see "Manipulating an HI Object's Size and Location" (page 4-47).

CalculateOptimalSize

Gets an HI object's optimal size.

IDL DECLARATION

```
void CalculateOptimalSize (
                    out SInt16 optimalWidth,
                    out SInt16 optimalHeight);
```

C DECLARATION

```
void HIObject_CalculateOptimalSize (
                    HIObject *somSelf,
                    Environment *ev,
                    SInt16 *optimalWidth,
                    SInt16 *optimalHeight);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`optimalWidth` A pointer to an integer. On output, the integer specifies the object's optimal width in pixels. If the object can't calculate an optimal width, `CalculateOptimalSize` returns the value of the object's original width in this parameter.

`optimalHeight` A pointer to an integer. On output, the integer specifies the object's optimal height in pixels. If the object can't calculate an optimal height, `CalculateOptimalSize` returns the value of the object's original height in this parameter.

DISCUSSION

An object's optimal size is represented by the smallest rectangle that can completely enclose its content. This is useful information for objects that can resize themselves to enclose their content in an optimal fashion. For example, static text panels may vary in length depending on the language in which they are displayed or due to text substitution.

`CalculateOptimalSize` doesn't resize the HI object. After you have determined its optimal size, use the `SetSize` method (page 4-53) to resize it.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

The `HIObject` implementation of `CalculateOptimalSize` simply returns the value of the object's original width and height in the `optimalWidth` and `optimalHeight` parameters. Any subclass for an object that can resize itself to enclose its content in an optimal fashion should override `CalculateOptimalSize` to determine that optimal size.

SEE ALSO

For an overview of related methods, see "Manipulating an HI Object's Size and Location" (page 4-47).

GetUpdateRect

Gets an HI object's update rectangle—that is, the rectangle that encloses the entire area in which the object can draw.

IDL DECLARATION

```
void GetUpdateRect (in HICoordinateSystem coordinate,
                    out Rect updateRect);
```

C DECLARATION

```
void HIObject_GetUpdateRect (
                    HIObject *somSelf,
                    Environment *ev,
                    HICoordinateSystem coordinate,
                    Rect *updateRect);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`coordinate` The coordinate system you want `GetUpdateRect` to use to specify the rectangle in the `updateRect` parameter. You identify the coordinate system with one of the values defined in the `HICoordinateSystem` enumeration (page 4-13).

`updateRect` A pointer to a rectangle. On output, the rectangle describes the object's update rectangle in the coordinate system specified by the `coordinate` parameter.

DISCUSSION

The rectangle returned in the `updateRect` parameter is at least as large as the bounding rectangle, and it can be larger to accommodate visual elements like focus indicators, which are drawn outside of the typical object's bounding

rectangle. An object's update rectangle can be no more than 7 pixels larger (per side) than its bounding rectangle.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

The `HIObject` implementation of `GetUpdateRect` simply returns the object's bounding rectangle. If you are creating a subclass for an HI object that draws some kind of adornment (for example, a user input focus ring or default push button ring), then your implementation should return an update rectangle that includes room for those adornments.

Typically, the implementation of an override gets the panel's bounding rectangle and calls `InsetRect` to adjust the rectangle to encompass the entire update area.

SEE ALSO

For an overview of related methods, see "Manipulating an HI Object's Size and Location" (page 4-47).

Enabling and Disabling an HI Object

Enable

Enables an HI object.

IDL DECLARATION

```
void Enable ();
```

C DECLARATION

```
void HIObject_Enable(HIObject *somSelf,
                     Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

An enabled object can receive events and handle them appropriately.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see “Execution Environments” (page 4-11).

OVERRIDE INFORMATION

Subclasses for objects that need to generate some side effect when they are enabled should override the `Enable` method. For example, many controls adjust their appearance after changing from a disabled state to an enabled state to indicate that they can accept mouse clicks.

SEE ALSO

To disable an HI object, use the `Disable` method (page 4-59). To get an object's enabled state, use the `IsEnabled` method (page 4-60).

Disable

Disables an HI object.

IDL DECLARATION

```
void Disable ();
```

C DECLARATION

```
void HIObject_Disable(
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

A disabled object does not accept events, and clients shouldn't dispatch events to it.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses for objects that need to generate some side effect when they are disabled should override the `Disable` method. For example, many controls adjust their appearance after changing from an enabled state to a disabled state to indicate that they can no longer accept mouse clicks.

SEE ALSO

To enable an HI object, use the `Enable` method (page 4-58). To get an object's enabled state, use the `IsEnabled` method (page 4-60).

IsEnabled

Returns an HI object's enabled state.

IDL DECLARATION

```
boolean IsEnabled ();
```

C DECLARATION

```
boolean HIObject_IsEnabled (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
function result	A Boolean value indicating the enabled state of the object. The value `true` indicates that the object is enabled; the value `false` indicates that the object is disabled.

DISCUSSION

An enabled object can receive events and handle them appropriately. A disabled object does not accept events, and clients shouldn't dispatch events to it.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `IsEnabled`.

SEE ALSO

To enable an object, use the `Enable` method (page 4-58). To disable an object, use the `Disable` method (page 4-59).

Getting and Setting an HI Object's Visibility

Show

Makes an HI object visible.

IDL DECLARATION

```
void Show ();
```

C DECLARATION

```
void HIObject_Show  (HIObject *somSelf,
                     Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

The `Show` method makes an invisible object visible. If the object is already visible, `Show` has no effect. Calling `Show` makes an HI object redraw itself by calling its own `Invalidate` method (page 4-102).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses that need to generate some side effect when the object is shown should override the `Show` function.

SEE ALSO

To make an object invisible, use the `Hide` method (page 4-63). To check whether an object is visible, use the `IsVisible` method (page 4-64).

Hide

Hides an HI object.

IDL DECLARATION

```
void Hide();
```

C DECLARATION

```
void HIObject_Hide  (HIObject *somSelf,
                     Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

The `Hide` method makes a visible object invisible by erasing and invalidating its rectangle, thus permitting other objects in the same area of the screen to redraw themselves. The object erases itself by calling its `Erase` method (page 4-100), which in turn calls the `EraseContent` method (page 4-117).

A hidden HI object doesn't receive events and can't take user input focus.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses that need to generate some side effect when the object is hidden should override the `Hide` function.

SEE ALSO

To make an object visible, use the `Show` method (page 4-62). To check whether an object is visible, use the `IsVisible` method (page 4-64).

IsVisible

Returns an HI object's visibility state.

IDL DECLARATION

```
boolean IsVisible ();
```

C DECLARATION

```
boolean HIObject_IsVisible (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
function result	A Boolean value indicating the visibility of the object. A value of `true` indicates that the object is visible; a value of `false` indicates that the object isn't visible.

DISCUSSION

An invisible object doesn't receive events and can't take user input focus.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `IsVisible`.

SEE ALSO

To make an object visible, use the `Show` method (page 4-62). To make an object invisible, use the `Hide` method (page 4-63).

Getting and Setting an HI Object's Title

GetTitle

Gets the image reference for an HI object's title.

IDL DECLARATION

```
OSStatus GetTitle (out HIImageRef title);
```

C DECLARATION

```
OSStatus HIObject_GetTitle (
                    HIObject *somSelf,
                    Environment *ev,
                    HIImageRef *title);
```

`somSelf`	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
`title`	A pointer to an image reference. On output, this reference identifies the object's title image; or, if the object has no title, this reference is set to `NULL`.
function result	Information about result codes will be provided with later developer releases.

DISCUSSION

An HI object's title is specified by an image reference. The image reference identifies a text object, an icon, a picture, a pattern, other kinds of images, or combinations of images.

The HI Imaging Objects class library provides object life cycle management for all image references. For a discussion of the way this works for image references, see `SetTitle` (page 4-67).

The image reference `GetTitle` provides in the title parameter is a clone of the image reference that identifies the HI object's title. Thus, a successful call to `GetTitle` increments the reference count for the title's image reference by 1. Therefore, every call to `GetTitle` should eventually be matched by a call to the HI imaging object method `ReleaseImage`.

After you get an image reference for an HI object's title, it's possible to change the image identified by its image reference using methods defined by the HI Imaging Objects class library.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetTitle`.

SEE ALSO

For an introduction to imaging objects, see "Imaging Objects" (page 1-40).

For a discussion of the life cycle management provided by the HI Objects class library, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

To set the image reference for an object's title, use the `SetTitle` method (page 4-67).

SetTitle

Sets an HI object's title.

IDL DECLARATION

```
OSStatus SetTitle   (in ConstHIImageRef title,
                     in HIAdoptionFlags disposalAdoptionFlags);
```

C DECLARATION

```
OSStatus HIObject_SetTitle (
                     HIObject *somSelf,
                     Environment *ev,
                     ConstHIImageRef title,
                     HIAdoptionFlags disposalAdoptionFlags);
```

`somSelf`	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
`title`	A reference to the imaging object you want to use to set the object's title. You use the static method `GetHIImagingObject`, defined by class `HIImagingObject`, to get an imaging object.
`disposalAdoptionFlags`	Reserved.
function result	Information about result codes will be provided with later developer releases.

DISCUSSION

Most HI objects can have a title, which is identified by an image reference. A title's image reference typically identifies a text image (which encapsulates a text object and a text strike), but you can also use icons, pictures, patterns, other kinds of images, or combinations of images as titles.

The HI Imaging Objects class library provides object life cycle management for all image references in much the same way that the HI Objects class library manages multiple references to HI objects. To set an HI object's title, you first use the HI Imaging Objects class library to construct an image reference for the text, icon, picture, or other image you want to set as the title. At this point, the reference count for the new image reference is 1. You then pass the image reference to `SetTitle`. `SetTitle` always clones the image reference in the `title` parameter, so the reference count is now 2.

If you call `Release` (page 4-35) on an HI object that has a title, and if that call to `Release` brings the HI object's reference count to 0, `Release` calls `Terminate` (page 4-37), and `Terminate` calls the HI imaging object method `ReleaseImage` to decrement the reference count for the title's image reference. If at that point the image reference's reference count reaches 0, `ReleaseImage` calls the HI imaging object method `DisposeImage`.

Thus, if you want the life of an image reference to coincide with the life of the HI object whose title it specifies, you should call `ReleaseImage` on the image reference right after you call `SetTitle`. In the example just discussed, this brings

reference count for the image reference back down to 1, so that a future call to the HI object's `Terminate` method will bring the count down to 0 and the image reference will be disposed of at the same time as the HI object.

If you want to keep the image reference around even after the HI object is disposed of, don't call `ReleaseImage` on the image reference after you call `SetTitle`. This allows the reference count for the image reference to remain at 1 even after any objects whose title it specifies have been disposed.

If you call `SetTitle` with an image reference different from that for the existing title, `SetTitle` releases the current image reference for the title and clones the new image reference in the usual manner.

If you call `SetTitle` with the same image reference as its current title, then the HI object assumes that its title image has changed. The object then invalidates itself and adjusts itself to the title content appropriately.

After you have set an object's title, it's possible to change it by manipulating the image identified by its image reference using methods defined by the HI Imaging Objects class library.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses that need to generate some side effect when the object's title is set or changes should override the `SetTitle` function.

SEE ALSO

For an introduction to imaging objects, see "Imaging Objects" (page 1-40).

For a discussion of the life cycle management provided by HIObject, see "Initializing, Saving, and Disposing of an Object" (page 4-25).

To get the image reference for an object's title, use the `GetTitle` method (page 4-65).

Event Handling

The Toolbox provides a table of default event handlers for the default Apple event dispatcher associated with each process. These handlers determine which window (if any) a standard Toolbox event should be directed toward and resend the event to that window's event dispatcher. The Toolbox also provides a table of default handlers for the dispatcher associated with each window. These handlers either handle the event directly or call a method provided by `HIWindow` that handles the event.

If the event affects the window's content area, the `HIWindow` method calls a root panel method for the event, which calls a subpanel method, and so on until the event reaches a subpanel that can handle it.

The accompanying document, *Apple Events in Mac OS 8,* introduces the Mac OS 8 event model. For an introduction to Toolbox event routing, see "Toolbox Event Routing" (page 2-3). For definitions of some of the standard events, see "Toolbox Events Reference" (page 3-3).

This section describes the methods defined by `HIObject` that handle the basic activation, navigation, and mouse events for which the Toolbox provides default handlers. These methods are typically called by handlers or by other methods; your application doesn't need to call them unless it overrides the default event handling or implements a subclass of `HIEmbeddingPanel`.

The `HandleActivate` (page 4-73), `HandleDeactivate` (page 4-75), and `HandleNavigation` (page 4-77) methods handle activation and navigation events.

The `HandleMouseDownInContent` (page 4-79), `HandleMouseMovedInContent` (page 4-73), `HandleMouseStoppedMovingInContent` (page 4-85), and `HandleMouseUpInContent` (page 4-87) methods handle mouse events that occur in a window's bounding rectangle.

Handlers installed in the window dispatcher for other events can either handle them directly or call the `HandleAppleEvent` method (page 4-71) on the window object, which calls the `HandleAppleEvent` method on the root panel, and so on until the event reaches a subpanel that can handle it.

Event-handling methods include a `reply` parameter for consistency with the Apple Event Manager, but they ignore this parameter unless the documentation specifies otherwise.

HandleAppleEvent

Handles a given Apple event.

IDL DECLARATION

```
OSStatus HandleAppleEvent (in AEEventClass eventClass,
                    in AEEventID eventID,
                    in AppleEvent theEvent,
                    in AppleEvent reply,
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
HIObject_HandleAppleEvent (HIObject *somSelf, Environment *ev,
                    AppleEvent *theEvent,
                    AEEventClass eventClass,
                    AEEventID eventID,
                    AppleEvent *reply);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`eventClass` The event class of the event to be handled.

`eventID` The event ID of the event to be handled.

`theEvent` A pointer to an Apple event. On input, you supply the event to be handled.

`reply` A pointer to a reply Apple event. On output, the reply Apple event contains the appropriate reply parameters, if any.

`handlerTableRef`
The handler table reference for the handler table containing the handler that received this event.

function result A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the event isn't handled by the object. More information about result codes will be provided with later developer releases.

DISCUSSION

`HandleAppleEvent` acts as a catchall for Apple events that don't have a corresponding method defined by the HI Objects class library. The client for `HandleAppleEvent` is typically an embedding panel that needs to send such an event to a subpanel instantiated from a custom subclass or a system service such as the Text Services Manager. See "Override Information" for more details.

The `eventClass` and `eventID` parameters are identical to the equivalent values in the Apple event specified by the `theEvent` parameter. This information is provided in parameter form for the convenience of the `HandleAppleEvent` implementation.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

If you are creating a subclass for objects that need to handle events other than the standard Toolbox events, you should override the `HandleAppleEvent` method and handle the events from within your subclass implementation.

In addition, the object's initialization functions should register its interest in that event by calling the `HIPanel` method `RegisterInterestInEvent` and specifying the event class and event ID. This ensures that the event will be passed through the window dispatcher's handler table to the `HandleAppleEvent` method for the window object, root panel, and successive subpanels until it reaches the object or objects that know about it.

More than one HI object can register interest in the same event. If the event is a geometry event, it gets sent to the interested panel, if any, in the location where the event occurred. If it is a broadcast event, the root panel sends it to all interested subpanels.

SEE ALSO

For an overview of related methods, see "Event Handling" (page 4-70).

For an introduction to the use of the `HandleAppleEvent` and `RegisterInterestInEvent` methods, see "Registering a Panel's Interest in an Event" (page 2-24).

HandleActivate

Handles a Window Activated event.

IDL DECLARATION

```
OSStatus HandleActivate (
                    in AppleEvent theEvent,
                    in AppleEvent reply
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
OSStatus HIObject_HandleActivate (
                    HIObject *somSelf,
                    Environment *ev,
                    AppleEvent *theEvent,
                    AppleEvent *reply
                    AEHandlerTableRef handlerTableRef);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`theEvent` A pointer to an Apple event. On input, you supply the Window Activated event to be handled.

`reply` A pointer to an Apple event. On input, you supply the reply Apple event to be returned by the object.

`handlerTableRef` The handler table reference for the handler table containing the handler that received this event.

function result A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the event isn't handled by the object. More information about result codes will be provided with later developer releases.

DISCUSSION

When a window dispatcher receives a Window Activated event, its default handler for that event calls the window's `HandleActivate` method, which calls the root panel's `HandleActivate` method, which calls all its subpanels' `HandleActivate` methods.

If you call the `HandleActivate` method on an HI object that is being reactivated, and if the object was enabled when it was deactivated, it reenables itself.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses that need to generate some side effect when the HI object is activated should override the `HandleActivate` function.

SEE ALSO

For an overview of related event-handling methods, see "Event Handling" (page 4-70).

For information about enabling and disabling, see "Enabling and Disabling an HI Object" (page 4-58).

For a description of the Window Activation event and its default handlers, see "Window Activated" (page 3-16).

HandleDeactivate

Handles a Window Deactivated event.

IDL DECLARATION

```
OSStatus HandleDeactivate (
                    in AppleEvent theEvent,
                    in AppleEvent reply
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
OSStatus HandleDeactivate (
                    HIObject *somSelf,
                    Environment *ev,
                    AppleEvent *theEvent,
                    AppleEvent *reply
                    AEHandlerTableRef handlerTableRef);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`theEvent` A pointer to an Apple event. On input, you supply the Window Deactivated event to be handled.

`reply` A pointer to an Apple event. On input, you supply the reply Apple event to be returned by the object.

`handlerTableRef`
The handler table reference for the handler table containing the handler that received this event.

function result A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the event isn't handled by the object. More information about result codes will be provided with later developer releases.

DISCUSSION

When a window dispatcher receives a Window Deactivated event, its default handler for that event calls the window's `HandleDeactivate` method, which calls the root panel's `HandleDeactivate` method, which calls all its subpanels' `HandleDeactivate` methods.

`HandleDeactivate` disables the object as well as deactivating it.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses that need to generate some side effect when the HI object is activated should override the `HandleDeactivate` function.

SEE ALSO

For an overview of related event-handling methods, see "Event Handling" (page 4-70).

For information about enabling and disabling, see "Enabling and Disabling an HI Object" (page 4-58).

For a description of the Window Deactivated event and its default handlers, see "Window Deactivated" (page 3-16).

HandleNavigation

Handles a Navigation event.

IDL DECLARATION

```
OSStatus HandleNavigation (
                    in SInt8 key,
                    in SInt8 keyCode,
                    in EventModifiers modifiers,
                    in AppleEvent theEvent,
                    in AppleEvent reply
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
OSStatus HIObject_HandleNavigation (
                    HIObject *somSelf,
                    Environment *ev,
                    SInt8 key,
                    SInt8 keyCode,
                    EventModifiers modifiers,
                    AppleEvent *theEvent,
                    AppleEvent *reply
                    AEHandlerTableRef handlerTableRef);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`key` Information about this parameter will be provided with later developer releases.

`keyCode` Information about this parameter will be provided with later developer releases.

`modifiers`	Information about this parameter will be provided with later developer releases.
`theEvent`	A pointer to an Apple event. On input, you supply the Navigation event to be handled.
`reply`	A pointer to an Apple event. On input, you supply the reply Apple event to be returned by the object.
`handlerTableRef`	The handler table reference for the handler table containing the handler that received this event.
function result	A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the event isn't handled by the object. More information about result codes will be provided with later developer releases.

DISCUSSION

When a process dispatcher receives a Navigation event, its default handler for that event forwards the event to the dispatcher for the window that currently has user input focus. When a window dispatcher receives a Navigation event, its default handler for that event calls the window's `HandleNavigation` method, which calls the root panel's `HandleNavigation` method, which calls the `HandleNavigation` method on the subpanel that currently has user input focus, and so on through the container hierarchy for HI objects with user focus until the event gets handled.

If the event isn't handled by any of the method calls, the default handler resends the event to the process dispatcher as a Virtual key event (if possible), which is in turn processed by the Text Services Manager and transformed into a text event. This allows the original event (for example, a Tab keypress) a chance to be processed by the same window dispatcher as a text event if it isn't handled as a Navigation event.

The `key`, `keycode`, and `modifiers` parameters are identical to the equivalent values in the Apple event specified by the `theEvent` parameter. This information is provided in parameter form for the convenience of the `HandleNavigation` implementation.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses for objects that can take user input focus and can react to navigation events should override `HandleNavigation`. The `HIObject` implementation of this method does nothing and returns `noErr`.

SEE ALSO

For an overview of related event-handling methods, see "Event Handling" (page 4-70).

For information about user input focus, see "Controlling User Input Focus" (page 4-89).

For an introduction to Navigation event routing, see "Navigation Events" (page 2-14).

HandleMouseDownInContent

Handles a Mouse Down event in an HI object's bounding rectangle.

IDL DECLARATION

```
OSStatus HandleMouseDownInContent (
                    in Point portLocalWhere,
                    in EventModifiers modifiers,
                    in AppleEvent theEvent,
                    in AppleEvent reply
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
OS Status HIObject_HandleMouseDownInContent (
                    HIObject *somSelf,
                    Environment *ev,
                    Point *portLocalWhere,
                    EventModifiers modifiers,
                    AppleEvent *theEvent,
                    AppleEvent *reply
                    AEHandlerTableRef handlerTableRef);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`portLocalWhere`
A pointer to a point. On input, you supply the point where the user clicked in coordinates local to the object's graphics port.

`modifiers` A set of flags defined by the `EventModifiers` enumeration indicating which keyboard modification keys were pressed at the time the event was generated. Examples of modification keys include the Shift, Command, Control, and Option keys.

`theEvent` A pointer to an Apple event. On input, you supply the Mouse Down event to be handled.

`reply` A pointer to an Apple event. On input, you supply the reply Apple event to be returned by the object.

`handlerTableRef`
The handler table reference for the handler table containing the handler that received this event.

function result A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the user has moved the pointer outside of the object's bounding rectangle or has pressed the mouse button while the pointer is within a part of the object that doesn't handle Mouse Down events. More information about result codes will be provided with later developer releases.

DISCUSSION

After determining that a Mouse Down event occurred within the window's content area, a window's `HandleMouseDown` method resends the event to the process dispatcher as a Mouse Down in Content event. The process dispatcher forwards the event to the window dispatcher, and a handler installed in the window dispatcher passes it to the window object's `HandleMouseDownInContent` method, which passes it to the root panel's `HandleMouseDownInContent` method.

If any subpanel's bounding rectangle contains the point specified by the `portLocalWhere` parameter, the root panel's `HandleMouseDownInContent` method calls the subpanel's `HandleMouseDownInContent` method, and so on until the event reaches a subpanel that can handle it.

The `portLocalWhere` and `modifiers` parameters are identical to the equivalent values in the Apple event specified by the `theEvent` parameter. This information is provided in parameter form for the convenience of the `HandleMouseDownInContent` implementation.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses for HI objects that need to track mouse movement while the mouse button is pressed must override `HandleMouseDownInContent`. The `HandleMouseDownInContent` implementation should create a filtered handler table, push it onto the process dispatcher's handler table stack, and call `AEReceive` to track the mouse until the button is released.

The Mouse Moved, Mouse Stopped Moving, and Mouse Up handlers associated with the filtered table should call the HI object's `HandleMouseMovedInContent`, `HandleMouseStoppedMovingInContent`, and `HandleMouseUpInContent` methods, respectively. The `HandleMouseUpInContent` method should force an exit from the call to `AEReceive`. This returns the flow of control to the `HandleMouseDownInContent` method, which should pop the filtered handler table and exit.

If the HI object accepts the Mouse Down event, `HandleMouseDownInContent` should return `noErr`. If the HI object doesn't accept the Mouse Down event, `HandleMouseDownInContent` should return `errAEEventNotHandled`.

Subclasses should also support drag-and-drop behavior where appropriate. If the Mouse Down event occurs in a selection within the object's content, the subclass implementation should initiate a drag. Information on how to do this will be provided with later developer releases.

SEE ALSO

For an overview of related event-handling methods, see "Event Handling" (page 4-70).

For an introduction to the default Toolbox routing for the Mouse Down in Content event, see "Default Geometric Event Routing" (page 2-6).

For more information about modal states, see the accompanying document *Apple Events in Mac OS 8.*

HandleMouseMovedInContent

Handles a Mouse Moved event in an HI object's bounding rectangle.

IDL DECLARATION

```
OSStatus HandleMouseMovedInContent (
                    in Point portLocalWhere,
                    in EventModifiers modifiers,
                    in AppleEvent theEvent,
                    in AppleEvent reply
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
OSStatus HIObject_HandleMouseMovedInContent (
                    HIObject *somSelf,
                    Environment *ev,
                    Point *portLocalWhere,
                    EventModifiers modifiers,
                    AppleEvent *theEvent,
                    AppleEvent *reply,
                    AEHandlerTableRef handlerTableRef);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`portLocalWhere`
A pointer to a point. On input, you supply the point the mouse moved to in coordinates local to the object's graphics port.

`modifiers` A set of flags defined by the `EventModifiers` enumeration indicating which keyboard modification keys were pressed at the time the event was generated. Examples of modification keys include the Shift, Command, Control, and Option keys.

`theEvent` A pointer to an Apple event. On input, you supply the Mouse Down event to be handled.

`reply` A pointer to an Apple event. On input, you supply the reply Apple event to be returned by the object.

`handlerTableRef`
The handler table reference for the handler table containing the handler that received this event.

function result A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the event isn't handled by the object. More information about result codes will be provided with later developer releases.

DISCUSSION

If the user moves the mouse while the mouse button is pressed and the pointer is within a window's content area, User Input Services sends a Mouse Moved event to the appropriate application's default process dispatcher. A Mouse Moved event handler installed in the process dispatcher by a `HandleMouseDownInContent` method (page 4-79) passes the event directly to the `HandleMouseMovedInContent` method for the HI object in which mouse movement has occurred.

The `portLocalWhere` and `modifiers` parameters are identical to the equivalent values in the Apple event specified by the `theEvent` parameter. This information is provided in parameter form for the convenience of the `HandleMouseMovedInContent` implementation.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses of HI object that support mouse interaction should override `HandleMouseMovedInContent`. See the override information for `HandleMouseDownInContent` (page 4-79) for details.

SEE ALSO

For an overview of related event-handling methods, see "Event Handling" (page 4-70).

For a description of the Mouse Moved event and its default handlers, see "Mouse Moved" (page 3-10).

HandleMouseStoppedMovingInContent

Handles a Mouse Stopped Moving event in an HI object's bounding rectangle.

IDL DECLARATION

```
OSStatus HandleMouseStoppedMovingInContent (
                    in Point portLocalWhere,
                    in EventModifiers modifiers,
                    in AppleEvent theEvent,
                    in AppleEvent reply
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
OSStatus HIObject_HandleMouseStoppedMovingInContent (
                    HIObject *somSelf,
                    Environment *ev,
                    Point *portLocalWhere,
                    EventModifiers modifiers,
                    AppleEvent *theEvent,
                    AppleEvent *reply,
                    AEHandlerTableRef handlerTableRef);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`portLocalWhere`
A pointer to a point. On input, you supply the point at which the mouse stopped moving in coordinates local to the object's graphics port.

`modifiers`	A set of flags defined by the `EventModifiers` enumeration indicating which keyboard modification keys were pressed at the time the event was generated. Examples of modification keys include the Shift, Command, Control, and Option keys.
`theEvent`	A pointer to an Apple event. On input, you supply the Mouse Stopped Moving event to be handled.
`reply`	A pointer to an Apple event. On input, you supply the reply Apple event to be returned by the object.
`handlerTableRef`	The handler table reference for the handler table containing the handler that received this event.
function result	A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the event isn't handled by the object. More information about result codes will be provided with later developer releases.

DISCUSSION

If the user stops moving the mouse while the mouse button is pressed and the pointer is within a window's content area, User Input Services sends a Mouse Stopped Moving event to the appropriate application's default process dispatcher. A Mouse Stopped Moving event handler installed in the process dispatcher by a `HandleMouseDownInContent` method (page 4-79) passes the event directly to the `HandleMouseStoppedMovingInContent` method for the HI object in which mouse movement has stopped.

The `portLocalWhere` and `modifiers` parameters are identical to the equivalent values in the Apple event specified by the `theEvent` parameter. This information is provided in parameter form for the convenience of the `HandleMouseStoppedMovingInContent` implementation.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses of HI object that support mouse interaction should override `HandleMouseStoppedMovingInContent`. See the override information for `HandleMouseDownInContent` (page 4-79) for details.

SEE ALSO

For an overview of related event-handling methods, see "Event Handling" (page 4-70).

For a description of the Mouse Stopped Moving event and its default handlers, see "Mouse Stopped Moving" (page 3-11).

HandleMouseUpInContent

Handles a Mouse Up event in an HI object's bounding rectangle.

IDL DECLARATION

```
OSStatus HandleMouseUpInContent (
                    in Point portLocalWhere,
                    in EventModifiers modifiers,
                    in AppleEvent theEvent,
                    in AppleEvent reply
                    in AEHandlerTableRef handlerTableRef);
```

C DECLARATION

```
HIObject_HandleMouseUpInContent (HIObject *somSelf, Environment *ev,
                    Point *portLocalWhere,
                    EventModifiers modifiers,
                    AppleEvent *theEvent,
                    AppleEvent *reply
                    AEHandlerTableRef handlerTableRef);
```

`somSelf`	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
`portLocalWhere`	A pointer to a point. On input, you supply the point at which the Mouse Up even occurred in coordinates local to the object's graphics port.
`modifiers`	A set of flags defined by the `EventModifiers` enumeration indicating which keyboard modification keys were pressed at the time the event was generated. Examples of modification keys include the Shift, Command, Control, and Option keys.
`theEvent`	A pointer to an Apple event. On input, you supply the Mouse Up event to be handled.
`reply`	A pointer to an Apple event. On input, you supply the reply Apple event to be returned by the object.
`handlerTableRef`	The handler table reference for the handler table containing the handler that received this event.
function result	A result code. The result code `noErr` indicates that the object handled the event. The result code `errAEEventNotHandled` indicates that the event isn't handled by the object. More information about result codes will be provided with later developer releases.

DISCUSSION

If the user releases the mouse button and the pointer is within a window's content area, User Input Services sends a Mouse Up event to the appropriate application's default process dispatcher. A Mouse Up event handler installed in the process dispatcher by a `HandleMouseDownInContent` method (page 4-79) passes the event directly to the `HandleMouseUpInContent` method for the HI object under the pointer when the user releases the mouse button.

The `portLocalWhere` and `modifiers` parameters are identical to the equivalent values in the Apple event specified by the `theEvent` parameter. This

information is provided in parameter form for the convenience of the `HandleMouseUpInContent` implementation.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses of HI object that support mouse interaction should override `HandleMouseUpInContent`. See the override information for `HandleMouseDownInContent` (page 4-79) for details.

SEE ALSO

For an overview of related event-handling methods, see "Event Handling" (page 4-70).

For a description of the Mouse Up event and its default handlers, see "Mouse Up" (page 3-8).

Controlling User Input Focus

Your application must maintain the current user input focus for the HI objects it creates. It receives some assistance in this effort from windows, which help route focused events, and from embedding panels, which perform simple user input focus arbitration for their subpanels.

Your application can provide more complex user input focus arbitration rules in two ways: by subclassing from `HIEmbeddingPanel` and overriding the user input focus arbitration methods, or by instantiating your panels outside of an embedding panel and manually transferring focus to the correct panel as necessary.

To assign user input to and release it for an object, use the `TakeUserInputFocus` (page 4-90) and `ReleaseUserInputFocus` (page 4-92) methods. The object redraws itself with the appropriate appearance after you call one of these methods.

Before calling `ReleaseUserInputFocus`, use the `CanReleaseUserInputFocus` method (page 4-94) to ensure that the object is in a state that permits it to release focus. For example, `CanReleaseUserInputFocus` doesn't return `true` for an editable text panel that accepts only numbers between 10 and 100 until the panel's text is valid.

To find out whether an object currently has user input focus, use the `HasUserInputFocus` method (page 4-93).

If you are implementing an HI object subclass, you may want to specify whether your object can accept user input focus when you first initialize the object. You can do so by calling the `GetUserInputFocusFlags` (page 4-97) and `SetUserInputFocusFlags` (page 4-96) methods.

TakeUserInputFocus

Assigns user input focus to an HI object.

IDL DECLARATION

```
void TakeUserInputFocus ();
```

C DECLARATION

```
void HIObject_TakeUserInputFocus (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

An HI object that can receive user input focus redraws itself with the appropriate focused appearance for the current theme after a call to `TakeUserInputFocus`. Embedding panels keep track of user input focus for their subpanels and call `TakeUserInputFocus` and `ReleaseUserInputFocus` as necessary to control it.

You don't need to call `TakeUserInputFocus` unless you are using panels outside of any kind of embedding panel, in which case your application must ensure that the appropriate panel has user input focus at any particular time.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses of HI object that can accept user input focus should override `TakeUserInputFocus`. The implementation of `TakeUserInputFocus` should call its inherited `TakeUserInputFocus` method to maintain the integrity of its internal user input focus state. When the HI object acquires focus, the implementation should adjust its appearance accordingly (usually with the aid of the Appearance Manager).

SEE ALSO

For an overview of related methods, see "Controlling User Input Focus" (page 4-89).

ReleaseUserInputFocus

Releases user input focus.

IDL DECLARATION

```
void ReleaseUserInputFocus();
```

C DECLARATION

```
void HIObject_ReleaseUserInputFocus (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

An HI object that can receive user input focus redraws itself without its focused appearance after a call to `ReleaseUserInputFocus`. Embedding panels keep track of user input focus for their subpanels and call `TakeUserInputFocus` and `ReleaseUserInputFocus` as necessary to control it.

You don't need to call `ReleaseUserInputFocus` unless you are using panels outside of any kind of embedding panel, in which case your application must ensure that the appropriate panel has user input focus at any particular time.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses of HI object that can accept user input focus should override `ReleaseUserInputFocus`. The implementation of `TakeUserInputFocus` should call its inherited `ReleaseUserInputFocus` to maintain the integrity of its internal user input focus state. When the HI object releases focus, the implementation should adjust its appearance accordingly (usually with the aid of the Appearance Manager).

SEE ALSO

For an overview of related methods, see "Controlling User Input Focus" (page 4-89).

HasUserInputFocus

Returns an HI object's user input focus state.

IDL DECLARATION

```
boolean HasUserInputFocus ();
```

C DECLARATION

```
boolean HIObject_HasUserInputFocus (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

function result A Boolean value indicating the object's current user input focus state. The value `true` indicates that the object has focus; the value `false` indicates that the object doesn't have focus.

DISCUSSION

User input focus state is maintained throughout the container hierarchy. If a push button inside an embedding panel inside a root panel inside a window has user input focus, then so do the embedding panel, the root panel, and the window.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `HasUserInputFocus`.

SEE ALSO

For an overview of related methods, see "Controlling User Input Focus" (page 4-89).

CanReleaseUserInputFocus

Indicates whether an HI object can release user input focus.

IDL DECLARATION

```
boolean CanReleaseUserInputFocus ();
```

C DECLARATION

```
boolean HIObject_CanReleaseUserInputFocus (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf`	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
function result	A Boolean value indicating whether the object can give up user input focus. The value `true` indicates that the object can give up focus; the value `false` indicates that the object can't give up focus.

DISCUSSION

The `HIObject` implementation of `CanReleaseUserInputFocus` always returns `true`. The `HIPanel` implementation calls `HIPanel::HasValidContent` to see whether the panel's content is in a valid state. If this is the case, `HIPanel::CanReleaseUserInputFocus` returns `true`; otherwise, it returns `false`.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Subclasses of `HIObject` that may not wish to release user input focus under unusual circumstances should override `CanReleaseUserInputFocus`.

SEE ALSO

For an overview of related methods, see "Controlling User Input Focus" (page 4-89).

SetUserInputFocusFlags

Sets an HI object's user input focus support flags.

IDL DECLARATION

```
void SetUserInputFocusFlags (
                    in HIUserInputFOcusFlags flags);
```

C DECLARATION

```
void SetUserInputFocusFlags (
                    HIObject *somSelf,
                    Environment *ev,
                    HIUserInputFocusFlags flags);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`flags` The object user flags you want to set for the object. You specify the flags with values defined in the `HIUserInputFocusFlags` enumeration (page 4-13).

DISCUSSION

`SetUserInputFocusFlags` changes the user input flags only—not the object's actual user input focus. The client is responsible for tracking user input focus and changing it as necessary.

You typically call this method during an object's initialization to specify its support for user input focus.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `SetUserInputFocusFlags`.

SEE ALSO

For an overview of related methods, see "Controlling User Input Focus" (page 4-89).

GetUserInputFocusFlags

Returns an HI object's user input focus support flags.

IDL DECLARATION

```
HIUserInputFocusFlags GetUserInputFocusFlags ();
```

C DECLARATION

```
HIUserInputFocusFlags HIObject_GetUserInputFocusFlags (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

function result The object's current user input flags. These are specified with values defined in the `HIUserInputFocusFlags` enumeration (page 4-14).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetUserInputFocusFlags`.

SEE ALSO

For an overview of related methods, see "Controlling User Input Focus" (page 4-89).

Imaging

To make an HI object draw or erase itself (either in its own window or in any specified graphics port), use the `Draw` method (page 4-99) or the `Erase` method (page 4-100).

To invalidate an HI object so that it will redraw itself according to its current drawing mode (for example, when it receives the next Update event), use the `Invalidate` method (page 4-102). To get or set an object's drawing mode, use the `GetDrawingMode` method (page 4-103) or the `SetDrawingMode` method (page 4-105).

To get or set an HI object's background pixel pattern, use the `GetBackgroundPattern` method (page 4-106) or the `SetBackgroundPattern` method (page 4-107).

Draw

Causes an HI object to draw itself in a specified color graphics port.

IDL DECLARATION

```
void Draw (in CGrafPtr whichPort,
                    in RgnHandle drawRgn);
```

C DECLARATION

```
void HIObject_Draw (
                    HIObject *somSelf,
                    Environment *ev,
                    CGrafPtr whichPort,
                    RgnHandle drawRgn);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`whichPort` A pointer to a `CGrafPort` record for the graphics port in which you want to draw the object. If the value of this parameter is `NULL`, the object draws itself in its own window's `CGrafPort`.

`drawRgn` A handle to the region of the object in which you want to draw. If the value of this parameter is `NULL`, the object draws the entire object.

DISCUSSION

The `Draw` method allows an object to draw itself to some other port as well as to its window. Although an object must always be located in a window, drawing to another port (for example, during printing or when drawing offscreen) may sometimes be desirable.

The `Draw` method doesn't do the actual drawing, but it sets up the environment to prepare the HI object to draw itself. For example, it preserves the current graphics port, then sets the port to the one specified by the `whichPort` parameter. Also, the `Draw` method adjusts the background erase pattern (if the panel has a background pattern). The protected method `DrawContent` (page 4-115) performs the actual drawing.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `Draw`. `Draw` calls the protected method `DrawContent` (page 4-115) to do the actual work of drawing. To change the way an object looks, create your own subclass and override `DrawContent`.

SEE ALSO

For an overview of related methods, see "Imaging" (page 4-98).

Erase

Causes an HI object to erase itself in a specified color graphics port.

IDL DECLARATION

```
void Erase (in CGrafPtr whichPort, in RgnHandle eraseRgn);
```

C DECLARATION

```
void HIObject_Erase (HIObject *somSelf,
                     Environment *ev,
                     CGrafPtr whichPort,
                     RgnHandle eraseRgn);
```

somSelf	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
ev	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
whichPort	A pointer to a `CGrafPort` record for the graphics port in which you want to erase the object. If the value of this parameter is `NULL`, the object erases itself in its own window's `CGrafPort`.
drawRgn	A handle to the region of the object that you want to erase. If the value of this parameter is `NULL`, the object erases all of itself.

DISCUSSION

The `Erase` method allows an HI object to erase itself in some other port as well as in its window. Although an HI object must always be located in a window, drawing and erasing to another port (for example, during printing or when drawing offscreen) may sometimes be desirable.

The Erase method doesn't do the actual erasing. Much like the `Draw` method (page 4-99), `Erase` sets up the environment so that the panel can erase itself, then calls `EraseContent`.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `Erase`. The `Erase` method calls the protected method `EraseContent` (page 4-117) to do the actual work of erasing. To change the way an object erases, create your own subclass and override `EraseContent`.

SEE ALSO

For an overview of related methods, see "Imaging" (page 4-98).

Invalidate

Forces an HI object to redraw itself according to its drawing mode.

IDL DECLARATION

```
void Invalidate ();
```

C DECLARATION

```
void HIObject_Invalidate (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

After a call to `Invalidate`, an object redraws itself immediately, after the next update event, or at some later time, depending on its current drawing mode. For example, the `SetValue` method for a control such as a slider calls the

control's `Invalidate` method so the control will redraw itself to reflect its new value.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

It's not usually necessary to override `Invalidate`. If you do, your implementation should call its inherited `Invalidate` method to maintain invalidation state integrity.

SEE ALSO

For descriptions of the possible drawing modes, see "Drawing Modes" (page 4-12).

To set an object's drawing mode, use the `SetDrawingMode` method (page 4-105).

For an overview of related methods, see "Imaging" (page 4-98).

GetDrawingMode

Gets an HI object's drawing mode.

IDL DECLARATION

```
HIDrawingMode GetDrawingMode ();
```

C DECLARATION

```
HIDrawingMode HIObject_GetDrawingMode (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf`	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
function result	The object's current drawing mode. The drawing mode is specified by one of the values defined in the `HIDrawingMode` enumeration (page 4-12).

DISCUSSION

An object's drawing mode determines whether a call to `Invalidate` (page 4-102) will cause the object to redraw itself immediately, after the next update event, or at some later time.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetDrawingMode`.

SEE ALSO

To set an object's drawing mode, use the `SetDrawingMode` method (page 4-105).

For an overview of related methods, see "Imaging" (page 4-98).

SetDrawingMode

Sets an HI object's drawing mode.

IDL DECLARATION

```
void SetDrawingMode (in HIDrawingMode drawingMode);
```

C DECLARATION

```
void HIObject_SetDrawingMode (
                    HIObject *somSelf,
                    Environment *ev,
                    HIDrawingMode drawingMode);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`drawingMode` The drawing mode to which you want set the object. You specify the drawing mode with one of the values defined in the `HIDrawingMode` enumeration (page 4-12).

DISCUSSION

An object's drawing mode determines whether a call to `Invalidate` (page 4-102) will cause the object to redraw itself immediately, after the next update event, or at some later time. By default, an HI object redraws itself when it receives the next Update event.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You should override `SetDrawingMode` only if your subclass doesn't support a particular drawing mode. For example, some HI objects (such as menus) can't afford to wait for an Update event. The implementations for their subclasses could override `SetDrawingMode` to turn a request for `kHIDrawNextUpdateEvent` to `kHIDrawImmediately`.

SEE ALSO

To get an object's drawing mode, use the `GetDrawingMode` method (page 4-103).

For an overview of related methods, see "Imaging" (page 4-98).

GetBackgroundPattern

Returns an HI object's background pattern.

IDL DECLARATION

```
PixPatHandle GetBackgroundPattern();
```

C DECLARATION

```
PixPatHandle HIObject_GetBackgroundPattern (
                    HIObject *somSelf, Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

function result A handle to a pixel pattern that describes the object's background pattern.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetBackgroundPattern`.

SEE ALSO

To set an HI object's background pattern, use the `SetBackgroundPattern` method (page 4-107).

For an overview of related methods, see "Imaging" (page 4-98).

SetBackgroundPattern

Sets an HI object's background pattern.

IDL DECLARATION

```
void SetBackgroundPattern (in PixPatHandle backgroundPattern);
```

C DECLARATION

```
void HIObject_SetBackgroundPattern (
                    HIObject *somSelf,
                    Environment *ev,
                    PixPatHandle backgroundPattern);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`backgroundPattern`
A handle to a pixel pattern. On input, you specify the pixel pattern to which you want to set the object's background. To get the current theme's background pattern for objects of a given class, use Appearance Manager functions.

DISCUSSION

The background pattern determines how both drawing and erasing operations draw the background for an HI object (such as a dialog box) that has one. You can set an object's background pattern to keep it consistent with the current theme.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

It's not usually necessary to override `SetBackgroundPattern`.

SEE ALSO

To get an object's background pattern, use the `GetBackgroundPattern` method (page 4-106).

For an overview of related methods, see "Imaging" (page 4-98).

Supporting Clipboard Operations

Some kinds of HI objects allow a user to perform Cut, Copy, Paste, Clear, or Undo operations while the object has user input focus. Your application should call the `Cut` (page 4-109), `Copy` (page 4-110), `Paste` (page 4-111), and `Clear` (page 4-113) methods for such objects as appropriate in response to the user's actions.

An HI object's ability to respond to these methods depends on its current state. You can use the `GetClipboardSupportFlags` method (page 4-114) to get flags that provide information about an object's current state with respect to the Clipboard.

For an overview of operations involving the Clipboard, see "Copy, Paste, Drag, and Drop" (page 1-42).

Cut

Cuts current selection.

IDL DECLARATION

```
void Cut ();
```

C DECLARATION

```
void HIObject_Cut   (HIObject *somSelf,
                     Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

The `HIObject` implementation of `Cut` simply calls the `Copy` (page 4-110) and `Clear` (page 4-113) methods.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You should never override the `Cut` method for standard cut operations. Instead, override `Copy` or `Clear`, and Cut will be supported automatically.

SEE ALSO

For an overview of related methods, see "Supporting Clipboard Operations" (page 4-109).

Copy

Copies current selection.

IDL DECLARATION

```
void Copy ();
```

C DECLARATION

```
void HIObject_Copy  (HIObject *somSelf,
                     Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

The `Copy` method uses the Scrap Manager and the Clipboard Manager to copy the HI object's current selection to the Clipboard.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You should override the `Copy` method for objects that are editable and support the Scrap Manager. Your implementation doesn't need to call its inherited `Copy` method.

SEE ALSO

For an overview of related methods, see "Supporting Clipboard Operations" (page 4-109).

Paste

Pastes at the location of the current selection.

IDL DECLARATION

```
void Paste();
```

C DECLARATION

```
void HIObject_Paste (HIObject *somSelf,
                     Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

The `Paste` method uses the Clipboard Manager and the Scrap Manager to copy the contents of the Clipboard to the HI object's current selection.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You should override the `Paste` method for objects that are editable and support the Scrap Manager. Your implementation doesn't need to call its inherited `Paste` method.

SEE ALSO

For an overview of related methods, see "Supporting Clipboard Operations" (page 4-109).

Clear

Clears the current selection.

IDL DECLARATION

```
void Clear ();
```

C DECLARATION

```
void HIObject_Clear (HIObject *somSelf,
                     Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

The `Clear` method clears the HI object's current selection.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

You should override the `Clear` method for objects that are editable and support the Scrap Manager. Your implementation doesn't need to call its inherited `Clear` method.

SEE ALSO

For an overview of related methods, see "Supporting Clipboard Operations" (page 4-109).

GetClipboardSupportFlags

Returns an HI object's clipboard support flags.

IDL DECLARATION

```
HIClipboardSupportFlags GetClipboardSuportFlags();
```

C DECLARATION

```
HIClipboardSupportFlags HIObject_GetClipboardSupportFlags (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

function result The object's clipboard support flags. These are specified with the values defined in the `HIClipboardSupportFlags` enumeration (page 4-14).

DISCUSSION

The clipboard support flags returned by `GetClipboardSupportFlags` provide information about an object's current state with respect to the Clipboard. You can use this information to determine whether it's appropriate to enable or disable menu items in the Edit menu while the object has user input focus.

Each HI object is responsible for keeping track of its own state with respect to Clipboard-related commands and for using the protected method `SetClipboardSupportFlags` (page 4-123) to set the flags that reflect that state.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

OVERRIDE INFORMATION

Do not override `GetClipboardSupportFlags`.

SEE ALSO

For an overview of related methods, see "Supporting Clipboard Operations" (page 4-109).

Protected Methods

Protected methods should be called only from within an HI object's implementation. Therefore, they are of interest only if you are creating a subclass.

DrawContent

Draws an HI object within its update rectangle.

IDL DECLARATION

```
void DrawContent    (in CGrafPtr whichPort,
                     in RgnHandle drawRgn);
```

C DECLARATION

```
void HIObject_DrawContent (
                    HIObject *somSelf,
                    Environment *ev,
                    CGrafPtr whichPort,
                    RgnHandle drawRgn);
```

somSelf	A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.
ev	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
whichPort	A pointer to a `CGrafPort` record for the graphics port in which you want to draw the object. If the value of this parameter is `NULL`, the object draws itself in its own window's `CGrafPort`.
drawRgn	A handle to the region of the object in which you want to draw. If the value of this parameter is `NULL`, the object draws the entire object.

DISCUSSION

`Draw` (page 4-99) calls `DrawContent` to draw an object within its update rectangle. `DrawContent` assumes that the graphics port and the object's background pattern have been set.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

The `DrawContent` method is protected—that is, it should be called only from within an object's implementation. When your application needs to draw an object, use the `Draw` method (page 4-99).

OVERRIDE INFORMATION

To ensure that its HI objects visually match the current theme, a subclass that overrides `DrawContent` shouldn't make any assumptions about the drawing context. In some cases, such as a visual separator panel, the subclass doesn't use QuickDraw primitives directly. Instead, it uses the Appearance Manager. For example, a visual separator panel calls the Appearance Manager function `DrawThemeSeparator` rather than the QuickDraw function `LineTo`. In other cases the subclass must obtain information (such as the current theme's background pattern) from the Appearance Manager before using QuickDraw functions.

EraseContent

Erases an HI object within its update rectangle.

IDL DECLARATION

```
void EraseContent (in CGrafPtr whichPort,
                   in RgnHandle drawRgn);
```

C DECLARATION

```
void HIObject_EraseContent (
                   HIObject *somSelf,
                   Environment *ev,
                   CGrafPtr whichPort,
                   RgnHandle drawRgn);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`whichPort` A pointer to a `CGrafPort` record for the graphics port in which you want to erase the object. If the value of this parameter is `NULL`, the object erases itself in its own window's `CGrafPort`.

`drawRgn`	A handle to the region of the object you want to erase. If the value of this parameter is `NULL`, the object erases all of itself.

DISCUSSION

`Erase` (page 4-100) calls `EraseContent` to erase a specified region of an object.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

The `EraseContent` method is protected—that is, it should be called only from within an object's implementation. When your application needs to erase an object, use the `Erase` method (page 4-100).

OVERRIDE INFORMATION

You can override the `EraseContent` method if your object draws in a restricted portion of its update rectangle and therefore needs to erase only the same restricted area. For example, a rectangular visual separator consists of four thin lines, not the entire update rectangle.

You can assume that the background pattern has been set before `EraseContent` is called.

TranslatePoint

Translates a given point between coordinate systems.

IDL DECLARATION

```
void TranslatePoint (in HICoordinateSystem translateFrom,
                     in HICoordinateSystem translateTo,
                     in Point sourcePoint,
                     out Point translatedPoint);
```

C DECLARATION

```
void HIObject_TranslatePoint (
                    HIObject *somSelf,
                    Environment *ev,
                    HICoordinateSystem translateFrom,
                    HICoordinateSystem translateTo,
                    Point *sourcePoint,
                    Point *translatedPoint);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`translateFrom` The coordinate system currently used to define the point in the `sourcePoint` parameter. You identify the coordinate system with one of the values defined in the `HICoordinateSystem` enumeration (page 4-13).

`translateTo` The coordinate system to which you want to translate the point in the `sourcePoint` parameter. You identify the coordinate system with one of the values defined in the `HICoordinateSystem` enumeration (page 4-13).

`sourcePoint` A pointer to a point. On input, you supply the point you want translated.

`translatedPoint`
A pointer to a point. On output, the point is a translation of the point in the `sourcePoint` parameter from the coordinate system described in the `translateFrom` parameter to the coordinate system described in the `translateTo` parameter.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see “Execution Environments” (page 4-11).

The `TranslatePoint` method is protected—that is, it should be called only from within an object's implementation. When your application needs to erase an object, use the `Erase` method (page 4-100).

OVERRIDE INFORMATION

Your subclass may introduce a new coordinate system that relates to its architecture. For example, class `HIScrollingPanel` provides a coordinate system for the content panel that is scrolled. If so, you should override `TranslatePoint` to support translating to and from your new coordinate system.

TranslateRect

Translates a given rectangle between coordinate systems.

IDL DECLARATION

```
void TranslateRect (in HICoordinateSystem translateFrom,
                    in HICoordinateSystem translateTo,
                    in Rect sourceRect,
                    out Rect translatedRect);
```

C DECLARATION

```
void HIObject_TranslateRect (
                    HIObject *somSelf,
                    Environment *ev,
                    HICoordinateSystem translateFrom,
                    HICoordinateSystem translateTo,
                    Rect *sourceRect,
                    Rect *translatedRect);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev`	A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.
`translateFrom`	The coordinate system currently used to define the rectangle in the `sourceRect` parameter. You identify the coordinate system with one of the values defined in the `HICoordinateSystem` enumeration (page 4-13).
`translateTo`	The coordinate system to which you want to translate the rectangle in the `sourceRect` parameter. You identify the coordinate system with one of the values defined in the `HICoordinateSystem` enumeration (page 4-13).
`sourceRect`	A pointer to a rectangle. On input, you supply the rectangle you want translated.
`translatedRect`	A pointer to a rectangle. On output, the rectangle is a translation of the rectangle in the `sourceRect` parameter from the coordinate system described in the `translateFrom` parameter to the coordinate system described in the `translateTo` parameter.

DISCUSSION

The `HIObject` implementation of `TranslateRect` simply calls `TranslatePoint` (page 4-118) on `TopLeft` and `BottomRight` of the rectangle.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

The `TranslateRect` method is protected—that is, it should be called only from within an object's implementation.

OVERRIDE INFORMATION

Do not override `TranslateRect`. Instead, override `TranslatePoint`.

StateChanged

Causes an HI object to invoke its state change functions.

IDL DECLARATION

```
void StateChanged   (in HIStateStateChangeCodeCreator selectorCreator,
                     in HIStateChangeCode changedCode);
```

C DECLARATION

```
void HIObject_StateChanged (
                     HIObject *somSelf,
                     Environment *ev,
                     HIStateStateChangeCodeCreator selectorCreator,
                     HIStateChangeCode changedCode);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`selectorCreator`
The creator code corresponding to the `changedCode` parameter.

`changedCode` The state change code that corresponds to the behavior you want to invoke with the state change function. The state change codes defined by class `HIObject` are listed in the `HIStateChangeCode` enumeration (page 4-16).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

The `StateChanged` method is protected—that is, it should be called only from within an object's implementation.

OVERRIDE INFORMATION

If your subclass needs to know whether the HI object's state has changed, you should override `StateChanged` and your implementation should call the inherited `StateChanged` method.

SetClipboardSupportFlags

Sets an HI object's clipboard support flags.

IDL DECLARATION

```
void SetClipboardSupportFlags (in HIClipboardSupportFlags flags);
```

C DECLARATION

```
void HIObject_SetClipboardSupportFlags (
                    HIObject *somSelf,
                    Environment *ev,
                    HIClipboardSupportFlags flags);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`flags` The clipboard support flags you want to set for the object. You specify these flags with values defined in the `HIClipboardSupportFlags` enumeration (page 4-14).

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

The `SetClipboardSupportFlags` method is protected—that is, it should be called only from within an object's implementation.

OVERRIDE INFORMATION

You can override `SetClipboardSupportFlags` if your subclass has special Clipboard-related requirements. For example, if the object you are defining can't ever paste, your implementation can ensure that the `kHISupportsPaste` flag is never set.

Verify

Verifies that an HI object's internal state is valid.

IDL DECLARATION

```
OSStatus Verify ();
```

C DECLARATION

```
OSStatus HIObject_Verify (
                    HIObject *somSelf,
                    Environment *ev);
```

`somSelf` A pointer to an HI object. All nonstatic SOM object methods take the SOM object as a first parameter. For details, see the SOM documentation provided by IBM.

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

DISCUSSION

This method is used for testing purposes only. In optimized builds, it should do nothing and return `noErr`. You should use this method in development builds of your application to facilitate debugging.

CALLING RESTRICTIONS

Methods defined by `HIObject` and its subclasses can be called only by the main task of a cooperative program. For more details, see "Execution Environments" (page 4-11).

The `Verify` method is protected—that is, it should be called only from within an object's implementation.

OVERRIDE INFORMATION

Subclasses should override this method so they can verify their own state.

Application-Defined Function

MyStateChangeCallback

Performs the action associated with an HI object's change in state.

```
void MyStateChangeCallback (
                    Environment *ev,
                    HIStateChangeCodeCreator selectorCreator,
                    HIStateChangeCode whatHappened,
                    HIObject *theObject);
```

`ev` A pointer to an environment variable used by all SOM classes to pass exception information back to the caller. For details, see the SOM documentation provided by IBM.

`selectorCreator`
: A state change creator code. This may be the value defined by class `HIObject` in the `HIStateChangeCodeCreator` enumeration (page 4-16) or your application's signature.

`whatHappened`
: A state change code. This may be one of the values defined by class `HIObject` in the `HIStateChangeCode` enumeration (page 4-16) or additional numerations defined by a subclass.

`theObject`
: A pointer to an HI object. This parameter identifies the object that invoked the `MyStateChangeCallback` function.

DISCUSSION

All state change functions associated with an HI object are called for all state changes that object experiences. Therefore, you must ensure that your state change function reacts only to those state change codes that it supports.

More information about state change callback functions will be available with later developer releases.

Notes for System 7 Developers

This appendix includes the general Toolbox compatibility guidelines that System 7 applications must follow to be able to run on Mac OS 8. It also compares the Mac OS 8 Scrap Manager, Clipboard Manager, Drag Manager, and Resource Manager with the equivalent System 7 services.

This appendix doesn't describe the interfaces that are entirely new in Mac OS 8, such as the Appearance Manager and the HI Objects class library. For an overview of the entire Mac OS 8 Toolbox, see "Introduction to the Mac OS 8 Toolbox" (page 1-3). For a detailed description of class `HIObject`, see "HIObject Class Reference" (page 4-5).

If you have developed a System 7 application and want to begin planning its migration to Mac OS 8, this appendix provides some of the information you need to get started. However, it doesn't include detailed information about the Mac OS 8 event model, resource formats, human interface guidelines, and other aspects of Mac OS 8 that you will also need to learn about.

For an introduction to the Mac OS 8 event model, see the accompanying document *Apple Events in Mac OS 8.*

For preliminary human interface guidelines, see the accompanying document *Human Interface Guidelines for Mac OS 8.*

▲ **WARNING**
This document is preliminary and incomplete. All information presented here is subject to change. ▲

Compatibility Guidelines

Using the Mac OS 8 Toolbox not only ensures compatibility with Mac OS 8 but also lays the foundation for new capabilities that will be introduced in Gershwin and future Mac OS enhancements.

Like any major system software revision, Mac OS 8 introduces features that aren't backward compatible with earlier systems. However, most System 7 applications can run on Mac OS 8, even though they may not be able to take advantage of all its features. For example, clients of standard System 7 definition procedures (defprocs) work correctly and inherit the Mac OS 8 human interface appearance. Custom defprocs written for System 7 also work correctly on Mac OS 8 but do not inherit the Mac OS 8 appearance.

Here are some guidelines you can use now to ensure that System 7 applications currently under development are compatible with the Mac OS 8 Toolbox:

- Support Apple events as described in *Inside Macintosh: Interapplication Communication*, including factoring your application and making it fully scriptable and recordable. The Mac OS 8 event model is based primarily on Apple events.
- Don't assume that dialog box backgrounds are white. The Mac OS 8 human interface supports a variety of background colors.
- For floating windows, use the standard floating window definition (ID 124) introduced in System 7.5. This window definition works correctly on Mac OS 8 and inherits the Mac OS 8 appearance.
- Don't hard-code any assumptions about the precise locations of human interface elements such as close boxes, zoom boxes, and window titles within the noncontent areas of windows or dialog boxes.
- Don't hard-code any assumptions about the precise locations of any human interface elements in the Save and Open dialog boxes. Use the relative position of the standard elements to determine the locations of new ones.
- Never access low memory directly. If you need to access low memory, use accessor functions.

- Use data structure accessor functions where they exist. For example, use `SetMenuItemText` and `GetMenuItemText` to manipulate menu item text rather than accessing the data structure directly.
- If data structure accessor functions aren't available, isolate the code that accesses data structures directly. Mac OS 8 provides accessor functions for all data structures, and it is easier to take advantage of them if you have isolated the code that needs to be updated.
- Don't manipulate the window list directly. Use the `BringToFront` and `SendBehind` functions instead.

Window Manager, Dialog Manager, Control Manager, List Manager, Menu Manager

The Window Manager, Dialog Manager, Control Manager, List Manager, and Menu Manager are supported for backward compatibility only. You can use the HI Objects class library to create windows, dialog boxes, alert boxes, controls, lists, and menus in Mac OS 8.

For an introduction to HI objects, see "Introduction to the Mac OS 8 Toolbox" (page 1-3).

Scrap Manager

The Scrap Manager used in System 7 has changed little since it was first created as part of the software for the original Macintosh computer. It was originally designed to handle a few lines of text or a 1-bit picture being copied and pasted between MacWrite and MacPaint, not the large pieces of data, such as QuickTime movies, sounds, and blocks of formatted text, commonly used today.

Mac OS 8 replaces the original Scrap Manager with a new Clipboard Manager and introduces an entirely new Scrap Manager. The new Scrap Manager supplies the generic storage mechanism for copying and pasting clipboard information and dragging and dropping data. Both the Clipboard Manager and the Drag Manager use the Scrap Manager to move data between clients (for

instance, between applications or within areas of a single application). The System 7 Drag Manager functions that prepared data for transport have been revised and incorporated into the Mac OS 8 Scrap Manager so that they apply to clipboard data as well as to drag information.

The Mac OS 8 Scrap Manager allows you to create a scrap, add items to the scrap, and specify each scrap item with different scrap item types (that is, multiple representations). It also allows you to read and extract information from a scrap after it has been transported to its destination.

Many Scrap Manager, Clipboard Manager, and Drag Manager functions take a scrap reference as an input parameter. A scrap reference identifies a particular scrap, whether it is used by the Clipboard Manager, the Drag Manager, or the Scrap Manager.

Scrap Manager Functions

The Mac OS 8 Scrap Manager provides functions you can use to create and delete scrap references, add items to the scrap, make and keep promises, and obtain information about scrap items.

Creating and Deleting Scrap References

You use the `NewScrapRef` function to create a new scrap and allocate a scrap reference for use with the Clipboard. The `DisposeScrapRef` function disposes of a scrap previously created by the `NewScrapRef` function.

Adding Scrap Items to the Scrap

The `AddScrapItemType` function, which replaces the System 7 Drag Manager function `AddDragItemFlavor`, lets you write data in a specific format to the scrap. You can use `AddScrapItemType` repeatedly to place data in more than one format in the scrap.

To add data to a specific item type, you can use the `SetScrapItemTypeData` function, which replaces the System 7 Drag Manager function `SetDragItemFlavorData`.

Every scrap item type has a set of attributes, which include information such as whether the scrap item type is private to the sender, whether the sender can translate the data, and so on. To set the current set of scrap item type attributes for a specified scrap, you can use the `SetScrapItemTypeAttributes` function.

Making and Keeping Promises

The Mac OS 8 Scrap Manager adds support for promises in clipboard operations to the support provided by System 7 for promises in drag operations. Your application makes a promise by adding a scrap item type with data of length 0 to a scrap item. When the user performs an action (such as choosing the Paste command or dragging and dropping) that requires a promise to be fulfilled, the Scrap Manager sends the Scrap Promise event to the application that made the promise to request that it fulfill its promise. You use Apple Event Manager functions to install your application's handlers for the Scrap Promise event.

Getting Scrap Item Information

The Mac OS 8 Scrap Manager provides functions that obtain a range of data about scrap items, including the number of scrap items and scrap item types, the scrap item reference for a specified scrap item, the number of item types for a specified scrap item, the scrap item type associated with a particular location within a scrap item, the size of a specified scrap item type, the attributes of a given scrap item type, and the data for a specified scrap item type. These scrap-item information functions replace the following System 7 Drag Manager functions: `CountDragItems`, `GetDragItemReferenceNumber`, `CountDragItemFlavors`, `GetFlavorType`, `GetFlavorDataSize`, and `GetFlavorData`.

Clipboard Manager

The entirely new Mac OS 8 Clipboard Manager supports the user's experience of cut, copy, and paste operations by accepting scraps created with the Scrap Manager and transferring them between clients via the familiar concept of the Clipboard.

Basically, the Mac OS 8 Clipboard contains a single Scrap Manager scrap. Clients use the Scrap Manager to create the scrap, the Clipboard Manager to put it on or retrieve it from the Clipboard, and the Scrap Manager to read it. The Clipboard Manager accepts a Scrap Manager scrap, returns read access to a scrap on the Clipboard, and releases the scrap when it's no longer needed.

The Clipboard Manager manages the Clipboard. When you use the Clipboard Manager to put a scrap on the Clipboard, you can no longer write to that scrap (except for unfulfilled promises). The Clipboard Manager automatically

disposes of the scrap when it has been replaced by a new scrap and other applications are finished retrieving it.

When using the Clipboard Manager to exchange data between applications or within your application, you follow these steps:

1. Create a scrap with the Scrap Manager's `NewScrapRef` function.
2. Add items to the scrap with the Scrap Manager's `AddScrapItemType` function.
3. Put the scrap on the Clipboard with the Clipboard Manager's `PutScrapOnClipboard` function.
4. When the user pastes, get the scrap from the Clipboard with the `GetClipboardScrapRef` function.
5. Obtain data from the scrap using the Scrap Manager functions.
6. Release the scrap with the `ReleaseClipboardScrap` function.

When a new scrap gets placed on the Clipboard, the Clipboard Manager sends all interested applications a Clipboard Changed event to notify them what data types are available for the new data. This allows each application to update its Edit menu appropriately as soon as new data is copied onto the Clipboard.

The data types specified in the Clipboard Changed event should be regarded as a hint, not a guarantee of the types that will actually be available on the Clipboard at some later time. For example, the user might choose the Paste command just before a new Clipboard Changed event arrives. Before handling a paste, your application should examine the Clipboard contents directly to check the available types at that time.

When the Clipboard Manager disposes of a scrap on the Clipboard that has been replaced by a new scrap, it sends all interested applications a Clipboard Disposed event. This informs an application that has added promises to the scrap that it no longer has any obligation to fulfill them.

You use Apple Event Manager functions to install your application's handlers for the Clipboard Changed and Clipboard Disposed events.

Clipboard Manager Functions

The Mac OS 8 Clipboard Manager functions let you place a scrap on the Clipboard and retrieve a scrap from it.

Putting a Scrap on the Clipboard

The `PutScrapOnClipboard` function takes a scrap created using the Scrap Manager and puts it on the Clipboard. If this function succeeds, the scrap becomes the active Clipboard scrap and the Clipboard Manager disposes of the scrap when it's finished with it. If the function fails, the client must dispose of the scrap.

Retrieving and Releasing a Scrap From the Clipboard

The `GetScrapFromClipboard` function retrieves a read-only copy of the scrap from the Clipboard. After retrieving a scrap from the Clipboard, you use Scrap Manager functions to extract its data. When you're finished with a scrap retrieved from the Clipboard, you use the `ReleaseClipboardScrap` function to release the scrap reference.

Drag Manager

The System 7 Drag Manager supports all aspects of drag-and-drop behavior, both in the Finder and within applications. The Mac OS 8 Drag Manager supports the user experience of dragging, but it no longer packs and unpacks the data that's transported in a drag. Instead, you use the Drag Manager to create the scrap, the Scrap Manager to add items to it, the Drag Manager to support the user experience during dragging, and the Scrap Manager to read the scrap after the drag operation is complete.

As mentioned in "Scrap Manager" (page A-3), the System 7 Drag Manager functions that prepare data for transport have been revised and incorporated into the Mac OS 8 Scrap Manager so that they apply to clipboard data as well as to drag information. The Mac OS 8 Drag Manager differs from the System 7 Drag Manager in several other important respects:

- As the user drags an image around the screen, the Mac OS 8 Drag Manager displays either a transparent version of the original image or just its outline.
- The Appearance Manager chooses the highlighting colors (for example, the target frame color) for drag operations.
- To add or obtain data from a drag, you use the Scrap Manager and pass the `DragScrapRef` data type directly.
- You no longer use the Drag Manager data type `FlavorFlags`. Instead, the Scrap Manager provides the equivalent type `ScrapItemTypeAttributeFlags`, used in collection items that you attach to a drag scrap using the Scrap Manager. Similarly, the System 7 data types `HFSFlavor` and `PromiseHFSFlavor` have been transferred to the Scrap Manager as `kScrapItemTypeFSObject` and `kScrapItemTypePromiseFSObject`.
- You use the Scrap Manager data type `ScrapItemType` instead of the System 7 data type `DragItemFlavor`.
- The System 7 Drag Manager `ItemReference` data type has been renamed `ScrapItemRef`, and the `FlavorType` data type has been renamed `ScrapItemType`. Their uses remain exactly the same.
- The following System 7 Drag Manager functions have been replaced by parallel functions in the Mac OS 8 Scrap Manager: `AddDragItemFlavor`, `SetDragItemFlavorData`, `SetDragSendProc`, `CountDragItems`, `GetDragItemReferenceNumber`, `CountDragItemFlavors`, `GetFlavorType`, `GetFlavorFlags`, `GetFlavorDataSize`, and `GetFlavorData`. These functions are supported for backward compatibility only.

When you use the Drag Manager to perform a drag operation, you follow these steps:

1. Create a drag scrap with the `NewDrag` function.
2. Add items to the scrap with the Scrap Manager's `AddScrapItemType` function.
3. Perform a single drag operation using the `TrackAEDrag` function. (This operation can be canceled.) The Drag Manager sends your application drag-tracking events to inform it of the progress of the drag.

4. When the user drops the dragged item, the Drag Manager sends your application a Drag Received event.
5. Use Scrap Manager functions in your Drag Received handler to retrieve data from the scrap.

Most Drag Manager functions take a drag scrap reference as an input parameter. A drag scrap reference can also be passed to Scrap Manager functions. A drag scrap reference must be allocated by the Drag Manager function `NewDrag` and is defined by the `DragScrapRef` data type.

Drag Manager Functions

The Mac OS 8 Drag Manager functions let you create and dispose of drag scrap references, override standard input and drawing behavior, perform a drag, set the transparent drag image, set and get information about a drag, and support drag-and-drop behavior.

Installing and Removing Drag Event Handlers

In Mac OS 8, the drag-tracking and drag receive handler functions used in System 7 have been replaced by Apple event handlers for equivalent drag events. The Drag Manager sends your application drag-tracking events to inform it of the progress of a drag and a Drag Received event when the user drops the dragged items. You use Apple Event Manager functions to install your application's handlers for these events.

Creating and Disposing of Drag References

The Mac OS 8 version of the `NewDrag` function creates a drag scrap reference to identify the drag in subsequent calls to the Drag Manager. This drag scrap reference is required when you add scrap item types via the Scrap Manager and when you call the `TrackAEDrag` function. Your installed drag event handlers receive this drag scrap reference, which they can use to call other Drag Manager functions.

The Mac OS 8 version of the `DisposeDrag` function disposes of the drag scrap identified by a specified drag scrap reference. If the drag scrap contains any scrap item types, the memory associated with the scrap item types is disposed of as well. You should call `DisposeDrag` after a drag has been performed using `TrackAEDrag` or to dispose of a drag scrap reference that's no longer needed.

Overriding Standard Drawing Behavior

The Mac OS 8 Drag Manager sends your application drag-drawing events to initiate the actual drawing during a drag. The Drag Manager provides default event handlers for these events, which your application can override if you wish to implement your own drag-drawing behavior.

Performing a Drag

Once the drag image for a drag has been set up, you use the Mac OS 8 version of the `TrackAEDrag` function to perform the drag operation with a particular drag scrap reference given a mouse-down event and drag region. The Drag Manager follows the cursor on the screen with the specified drag image feedback and sends drag-tracking events to inform your application of the progress of the drag. When the user releases the mouse button, the Drag Manager sends a Drag Received event to the destination window. Your application's Drag Received handler accepts the drag and transfers the dragged data into the application.

Setting the Transparency of the Drag Image

You can use the new `SetDragImage` function to set the degree of transparency of a given drag image.

Supporting Drag-and-Drop Behavior

Like the System 7 Drag Manager, the Mac OS 8 Drag Manager supports drag-and-drop behavior with functions that retrieve and set an Apple event descriptor for a specified drop location, perform standard drag-and-drop highlighting (including scrolling preparation), and let you draw zooming animation like the Finder's.

Getting and Setting Status Information About a Drag

The Mac OS 8 Drag Manager includes functions that obtain status information about the drag attribute flags, get and set the pointer location, retrieve the origin of a specified drag, obtain key modifiers associated with a drag scrap reference, and set and retrieve the bounding rectangle of drag items.

Resource Manager

Most of the System 7 Resource Manager functions are supported for backward compatibility. The exceptions are the `InitResources`, `RsrcZoneInit`, and `RsrcMapEntry` functions, which even System 7 applications don't need to call. Also, the undocumented resource chain override mechanism used by some System 7 applications is not supported in Mac OS 8.

Most of the System 7 functions also have an equivalent function, whose name begins with the prefix `RM`, in Mac OS 8. Major differences between the new functions and System 7 functions include the following:

- You can't access the resource map in Mac OS 8. Mac OS 8 supports an opaque resource file abstraction, and you access its resources through this abstraction.
- The error mechanism based on `ResError` is no longer necessary. Instead, Mac OS 8 Resource Manager functions simply return `OSStatus` errors.
- The `FSpCreateResFile`, `HCreateResFile`, and `CreateResFile` functions have been replaced by the single function `RMCreateResFile`, which takes a Mac OS 8 file system object.
- The `FSpOpenResFile`, `HOpenResFile`, `OpenRFPerm`, and `OpenResFile` functions have been replaced by the single function `RMOpenResFile`, which takes a Mac OS 8 file system object.
- The new functions `RMAddResFileToSearchPath` and `RMRemoveResFileFromSearchPath` allow you to remove resource files from the resource search path for your application and to add previously removed files to the beginning of the resource search path.
- Functions such as `Get1Resource` and `Get1NamedResource` are no longer needed. Instead, you specify parameters for `RMGetResource`, `RMGetNamedResource`, and so on that indicate whether or not the function should search just the current resource file or the entire resource search path.
- Mac OS 8 doesn't support ROM-based resources, so there is no Mac OS 8 equivalent to the `RGetResource` function.

- The new Resource Manager functions include a parameter that allows you to enable or disable automatic loading of resource data into memory. As a result, there is no need for a Mac OS 8 equivalent of the `SetResLoad` function.
- The System 7 function `GetResourceSizeOnDisk` has been renamed `RMGetResourceSize`.
- There is no Mac OS 8 equivalent to the `GetMaxResourceSize` function, because using it depends on specific implementation details of the System 7 Resource Manager.
- The `GetResFileAttrs` and `SetResFileAttrs` functions have been replaced by the `GetResFileReadOnlyState` and `SetResFileReadOnlyState` functions, which get and set the resource file's read-only state both in memory and on disk.

Glossary

active window The frontmost nonfloating window that is currently receiving user input. The active window is identified by distinctive details that aren't visible for inactive windows; for example, the Apple Default theme displays title bars for active windows with characteristic "racing stripes."

alert box A panel that an application displays 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. See **movable alert box, nonmovable alert box, dialog box, modal window.**

Apple event A data structure that identifies itself by event class and event ID, names its own destination, and contains additional data structures that vary depending on the kind of event. First introduced in System 7 to support interapplication communication, Apple events provide the primary mechanism for communication throughout Mac OS 8.

Apple event dispatcher A dispatcher that consists of an event queue and a handler table stack. For example, every process has a default Apple event dispatcher associated with at least one task, and a program may install additional dispatchers in a process as necessary. See also **handler table stack.**

application handler table A handler table added to a handler table stack by an application. See also **default handler table, handler table stack.**

background processing Processing that takes place in the background while the user continues to work. You perform background processing by creating an additional task that executes preemptively and concurrently while the main task that controls the human interface continues to respond to user actions.

bounding rectangle The rectangle that defines an HI object's location for the purposes of drawing its content or responding to geometric events. Compare **update rectangle.**

broadcast events Apple events that are routed to multiple targets. For example, a Window Activated event triggers the `HandleActivate` method for every panel within a window.

cell A rectangular part of a list displaying information about one item in the list.

Clipboard A container maintained by the Clipboard Manager that holds a shared Scrap Manager scrap.

close box The element in a window's title bar that, when clicked, closes the window. In the Apple Default theme, the close box is a box on the left end of the title bar of an active window.

collapse box The element in a window's title bar that, when clicked, collapses or expands the window. In the Apple Default theme, the collapse box is a box on the right end of the title bar of an active window.

composite imaging object An imaging object that can draw a composite image, identified by a single image reference, that consists of several different images. See also **imaging object.**

containment hierarchy A hierarchical arrangement of objects that describes which objects are contained by which other objects. Compare **inheritance hierarchy.**

data fork Part of a file that contains data accessed using the File Manager. The data 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 handler table A table of default handlers at the bottom of every handler table stack. See also **application handler table, handler table stack.**

dialog box A panel that an application displays to solicit specific kinds of information from the user. See also **modeless dialog box, movable modal dialog box, nonmovable modal dialog box.**

document window A window typically used to display document data. A document window appears behind floating and modal windows in an application's layer. See also **floating window, modal window.**

drag To position the pointer on a visual interface element (such as an icon in the Finder), press and hold the mouse button, move the pointer to a new position, and then release the mouse button. Dragging can have different effects, depending on what's under the pointer when the user first presses the mouse button. To support the dragging of items from one place to another, you use the Drag Manager.

drag region The portion of a window's title bar and, depending on the current theme, additional portions of the window frame that the user can drag to move a window.

embedding panel A panel of any subclass of `HIEmbeddingPanel` that contains other panels and controls their layout, user input focus, and navigation.

encapsulation In object-oriented programming, the packaging of an object's data and the functions that can act on it in a manner that protects the data from inappropriate changes. This protection is possible because only the object itself can change its data. To gain access to an object's data, a client must call that object's programming interface.

factoring Using Apple events to separate the code that presents an application's human interface to the user from the code that responds to the user's manipulation of the interface. In a fully factored application, any significant user actions generate Apple events that any scripting component based on the Open Scripting Architecture (OSA) can record as statements in a compiled script.

file A named, ordered collection of information stored on a Mac OS volume, typically divided into a data fork and a resource fork.

filtered handler table A handler table used to suspend the incoming events for which it doesn't provide handlers. When a filtered table contains no handler for a particular event, the event remains in the event queue without being handled. After the filtered table has been removed from the handler table stack, the Apple Event Manager passes any suspended events on to the next handler table in the order in which they were originally received. See also **nonfiltered handler table.**

floating window A window typically used for tool palettes, catalogs, and other elements used to act on data in document windows. A floating window appears in front of document windows and behind modal windows in an application's layer. See also **document window, modal window.**

focused events Apple events that are routed to a target that currently has user input focus. For example, text events are typically routed to the editable text panel currently receiving user input.

geometric events Apple events that are routed to a single target whose bounding rectangle contains coordinates specified by the event. For example, a Mouse Down event is typically directed to a single HI object, such as a button, that the user has clicked.

handler table A table of Apple event handlers that can be added to a handler table stack. Every handler table stack includes a default handler table, and an application can push additional handler tables onto the stack as necessary to express interest in specific Apple events. See also **application handler table, default handler table.**

handler table stack A stack of handler tables that consists of a default handler table and potentially one or more application handler tables. The Apple Event Manager looks through a dispatcher's handler table stack to find handlers for incoming events. See also **application handler table, default handler table.**

HI object See **human interface object.**

human interface object A SOM object created from a subclass of `HIObject` that encapsulates one or more human interface elements, such as a window, a dialog box, a control, or a menu.

Human Interface Toolbox A collection of shared libraries that application developers use to create and manipulate human interface elements such as windows, dialog boxes, menus, and controls. These shared libraries ensure that Mac OS 8 applications present a consistent and standard interface to users.

image reference A reference that you can pass to imaging object methods or HI object methods to identify an image to be drawn by an imaging object. See also **imaging object.**

imaging object A SOM object instantiated from a subclass of `HIImagingObject` that can draw a specific kind of image data, such as text, icons, or pictures. See also **image reference.**

inheritance In object-oriented programming, the transmission of properties and behaviors from one class to another. See also **inheritance hierarchy.**

inheritance hierarchy In object-oriented programming, an hierarchical arrangement of classes that describes their patterns of inheritance. Compare **containment hierarchy.**

list A series of items displayed within a rectangle.

mixed state A state in which a radio button or a checkbox can be displayed. A mixed state indicates a setting is on for some elements in a selection and off for others. The user can change a checkbox in a mixed state to either on or off for all the elements concerned, but can't directly change a checkbox that's on or off to a mixed state.

modal window A window that puts the user in a state or "mode" of being able to work only inside the window. A modal window is typically used to display a dialog box or alert box that requires immediate attention from the user. A modal window appears in front of all other windows in an application's layer. See also **document window, floating window.**

modeless dialog box A dialog box displayed in a document window without a size 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 **nonmovable modal dialog box, movable modal dialog box.**

movable alert box An alert box with a title bar that allows the user to move the box.

movable modal dialog box A dialog box displayed in a modal window with a title bar (but no close box) that the user can drag to move the box. The user can dismiss a movable modal dialog box only by clicking its buttons. See also **dialog box, nonmovable modal dialog box, modeless dialog box.**

nonfiltered handler table A handler table that allows the Apple Event Manager to process events as they arrive, without suspending any of them. When a nonfiltered handler table contains no handler for a particular event, the Apple Event Manager passes the event on to the next handler table in the stack. See also **filtered handler table.**

nonmovable alert box An alert box that the user can't move.

nonmovable modal dialog box A dialog box displayed in a modal window that the user can't move. The user can dismiss a nonmovable modal dialog box only by clicking its buttons. See also **modeless dialog box, movable modal dialog box.**

panel In the HI Objects class library, any HI object that can be placed in a window. Compare **window.**

periodic processing Processing that takes place at specified intervals. For example, if the user isn't doing anything else, an application should be able to perform repetitive tasks such as making the caret blink in the active window. To support periodic processing in Mac OS 8, you can use periodic Apple events.

polymorphism In object-oriented programming, the ability to call objects of different classes with the same method; for example, you can call the `Draw` method to draw an HI object of any class.

process dispatcher An Apple event dispatcher associated with a process.

promise A scrap item type that contains a placeholder rather than the scrap item data itself. When the user pastes or completes a drag operation, the Scrap Manager sends a Scrap Promise event to the original application to request that it fulfill its promise for the type of data being pasted or dropped. See also **scrap, scrap item type.**

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 Part of a file that contains the file's 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.

root panel An embedding panel that fills a window's content area and to which the window passes all events that affect the window's content. The root panel in turn passes events to other panels that it contains.

scrap A structure created by the Scrap Manager that consists of one or more scrap items, which in turn can hold one or more pieces of data. See also **promise, scrap item, scrap item type.**

scrap item A structure created by the Scrap Manager for holding a single piece of data that can be represented in different ways by one or more scrap item types. See also **scrap, scrap item type.**

scrap item type A structure created by the Scrap Manager that is associated with a scrap item and holds a single representation of the piece of data associated with the scrap item. See also **scrap, scrap item.**

size box A box in the lower-right corner of some active windows. Dragging the size box resizes the window.

SOM See **System Object Model.**

System Object Model (SOM) A standard architecture licensed by IBM for the development and packaging of object-oriented software. SOM provides language- and platform-independent means of defining programmatic objects and handling method dispatching dynamically at runtime.

theme A coordinated set of human interface designs that determine the appearance of human interface elements on a systemwide basis.

title bar icon An icon in a document window's title bar that users can use as a proxy for the document's Finder icon in drag-and-drop operations. For example, the user can drag a document's title bar icon to another volume, then drop it to copy a document file to that volume.

update rectangle The rectangle that defines the entire area in which an HI object can draw, including its bounding rectangle and any adornments. Compare **bounding rectangle.**

user input focus The current focal point on the screen for user input, whether from a pointing device, a keyboard, a speech input device, or other input devices.

window An object that presents information such as a document or message. In the HI objects class library, a window is also a container for all other kinds of HI objects, including menus and dialog boxes. Compare **panel.**

window dispatcher An Apple event dispatcher associated with a window.

window group A group of windows associated with a window. Whenever the user activates a window that has an associated window group, all the windows in the group also come as far forward as they can while maintaining their current ordering.

zoom box A box to the left of the collapse box in a window's title bar that the user can click to alternate between two different window sizes. Clicking the zoom box once causes the window to expand to its optimal size on the monitor on which most of its area is currently displayed. Clicking the zoom box a second time restores the window to its previous size and location. See also **collapse box.**