Upgrading to the Mac OS X HIToolbox

Carbon > Human Interface Toolbox

2004-06-28

Contents

Figures, Tables, and Listings

Introduction to Upgrading to the Mac OS X HIToolbox

One of the strengths of Mac OS X is its support of older Mac OS APIs, many of which were introduced before Mac OS 8, and some before System 7. This support made it easier to move older applications to Mac OS X. However, because many of these APIs were designed for a different system architecture, performance can suffer, and they often cannot take advantage of key Mac OS X features. Updating your application's user interface code to use the windowing and event-handling models designed specifically for Mac OS X gives immediate benefits in performance, code maintainablity, and portabiity, and it enables you to adopt the latest Mac OS X technologies.

Who Should Read This Document

This document is for Carbon developers who have older applications that would benefit from using modern HIToolbox technologies. You should consider upgrading if your application uses any of the following older Apple technologies:

- Resource-based windows, controls, or menus
- The Dialog Manager
- Procedure pointer–based custom control definitions
- QuickDraw
- Event handling using `WaitNextEvent`, `StillDown`, `WaitMouseUp`, or `Button`.

An updated application will use nib-based windows and menus, with HIView-based controls (both standard and custom). All event handling is done through Carbon events, and all drawing in views is done using Quartz.

HIViews and compositing mode are supported in Mac OS X v10.2 and later. Carbon events (including Carbon event–based control definitions) and nib files are supported in Mac OS X v10.0 and later.

Even if you do not want to adopt all these technologies in your application right away, this document can guide you towards incremental changes that will make it easier to adopt new features.

Organization of This Document

This document is organized into the following chapters:

- "The Modern Advantage" (page 9) describes the benefits of adopting the modern HIToolbox in your application.
- "Porting Steps" (page 13) describes steps you should take to port your application to use the modern HIToolbox.

- "A Porting Example: Converting a User Item to a Custom View" (page 33) shows how you could convert an old dialog user item to a custom HIView.

See Also

In addition to this document, you may find the following documents useful:

- Introducing HIView describes the concepts behind HIViews and explains how they differ from older user interface elements. It also contains numerous programming examples.
- Technical Note TN2074, HIView APIs Vs. Control Manager APIs, clarifies the differences and similarities between HIView and Control Manager functions.
- *Carbon Event Manager Programming Guide* provides an overview of the Carbon Event Manager and event handling in general for Carbon applications.
- If you have not yet upgraded your application to Carbon and Mac OS X, the *Carbon Porting Guide* guides you through the steps required. It also contains information about how to move to the Carbon Event Manager from the classic `WaitNextEvent` model.

The Modern Advantage

If your Carbon user interface still uses classic Mac OS technologies—such as resources, the Dialog Manager, `WaitNextEvent`, QuickDraw, and procedure pointer–based custom window, menu, or control definitions—Apple encourages you to upgrade your code to use their modern Mac OS X replacements.

- Nib files are the modern replacement for resource files in creating and managing controls, windows, and menus.
- HIView is the object-oriented view system for implementing Carbon user interface elements in Mac OS X.
- Carbon events is the modern event messaging system for Carbon applications.
- Quartz is the modern drawing API in Mac OS X.

This chapter outlines the advantages of using these technologies.

Nibs Replace Resources

Interface Builder provides an intuitive drag-and-drop method of laying out your user interface. It stores information about your interface in special nib files (or simply "nibs") that you can then load into your Carbon application. It also allows you to set commonly-used values such as control/view IDs, window attributes, initial values, and so on. Because the nib file is independent of your executable file, you can make changes to your user interface without having to recompile any code. For example, you can reposition controls or adjust initial radio button or checkbox states by simply modifying the appropriate nib file.

Wait No More

Carbon events replaces the classic `WaitNextEvent` event handling model. It is both simpler to use and more sophisticated than its predecessor, and it offers improved performance.

Carbon events eliminate the need for polling; your application no longer has to query the system for events. Instead, you simply register for the events you want to receive. These events get dispatched directly to the relevant target where individual event handlers can then process the event.

With this event model, you no longer need boilerplate code in your application to obtain and dispatch events. In addition, you can replace many functions that polled the system with Carbon event handlers. Some examples:

- Instead of using `Button` to determine if the user pressed the mouse button, you simply register to be notified whenever the mouse button is pressed.

- In the past, if you wanted to implement a timer, you had to poll the system to determine how many ticks had passed. Now, you can simply install a Carbon event timer that calls a handler when the specified time has elapsed.
- If you wanted to change window elements as the mouse rolled over them, you used to have to poll for the mouse position. Now, you can register mouse tracking areas with the system, and you are notified with a Carbon event whenever the mouse enters or leaves a given area.

In addition, if you use the standard window handler, the Carbon Event Manager installs numerous default handlers, most of which do what you would have wanted to do anyway. For example, in a window containing multiple HIViews, pressing Tab automatically advances the keyboard focus. You also benefit from Apple updates. For example, if Apple improved the algorithm for determining how to advance the focus, your application would gain this benefit automatically.

The standard event handlers also make it easy for your application to support assistive devices, as mandated by section 508 of the Workforce Investment Act of 1998. If your application uses standard user interface elements, you gain this support automatically.

Better Performance Through Compositing

Compositing mode (which you should not confuse with Quartz compositing) is a special window setting that attempts to minimize the amount of necessary drawing; images are redrawn only when they change, and areas that are hidden beneath opaque views are never drawn. Compositing mode is available in Mac OS X v10.2 and later.

The Mac OS X HIToolbox uses the term HIView to refer to controls that have been modified to optionally or exclusively support compositing mode. At heart, an HIView is nothing more than a new name for the existing `ControlRef` type. The `HIViewRef` type is identical to the `ControlRef` type , both at the programming language level and the implementation level. You can use most standard Control Manager calls on them, as well as the new `HIViewXXX` functions.

For example, the pushbutton control is considered an HIView, because it can be used in either a compositing or non-compositing window. The ListBox control is not considered an HIView, because it cannot be used in a compositing window. The Scroll view, introduced in Mac OS X 10.2, is considered an HIView, and can be used only in compositing mode. The difference between a control and a view, therefore, is primarily one of implementation changes that allow the control to behave correctly in a compositing window.

Compositing mode results in a number of performance advantages. For example, implementing live window resizing or scrolling in the past often resulted in images and controls being redrawn multiple times. At best, this resulted in sluggish performance, and at worst there may have been flickering onscreen due to the multiple redraws. In contrast, when in compositing mode, drawing occurs only once during a given pass through the event loop, and no image or control is drawn more often than necessary. To gain the benefits of compositing mode, all drawing in your window must be done from within a compositing-aware view (either a standard view or a custom view that you have written). Classic window update events are not provided for compositing windows.

In addition, compositing mode respects the z-order (that is, the layering) of controls, which makes it much easier to manipulate controls or views independently of each other. You no longer have to worry about erasing backgrounds, keeping track of control hierarchies, or aligning patterns when drawing or redrawing controls. For example, if you partially cover one view with another, you do not need to keep track of which view is above which, or worry about what happens when a view redraws.

New View-Based Controls

The HIView model introduces a number of new controls. Some of them simplify the work required to implement existing controls, while others provide new functionality altogether. Most work in both compositing and noncompositing windows.

- The combo box combines an editable text field and a pop-up menu.
- The scroll view automatically handles scrolling and live updating of scrollable information. By adopting this view, you no longer need to write code to calculate the positions of scroll thumbs or keep track of the content position. Note that the scroll view is available to windows only in compositing mode.
- The search field lets users enter search information, just as in Mail or Safari.
- The text view is a text container that uses the Multilingual Text Engine (MLTE). If you have only simple text-manipulation requirements, this view does most of the work for you.
- The web view provides an easy way to display HTML web content.
- The image view is a container for holding any sort of image.
- The segmented view is a segmented button much like the icon/list/column view switcher in the Finder.

In addition, if you want to create custom toolbar items in a toolbar, you must implement the items as custom HIViews.

The object-oriented nature of views makes it easier to customize; you can subclass or modify existing controls without too much effort, and the resulting code is much more portable. A custom view-based control may only require you to implement a few event handlers to override the behavior of a standard control. Another advantage to moving code "under the hood" is that when Apple fine-tunes, adds features, or otherwise improves functionality, your application can benefit automatically. You don't have to worry about retuning your code to adopt every change, and there's less likelihood of something breaking your code.

Draw With Quartz

Quartz is the imaging model for Mac OS X. In terms of features and flexibility, it is far superior to QuickDraw. New Apple technologies will leverage the power of Quartz, so it's in your best interest to make your application Quartz-savvy. Because HIView user interface objects were designed with Quartz in mind, you can easily use all its features.

Note that even though Quartz relies on a different coordinate system than QuickDraw, the system flips the context before presenting it to your HIView drawing handler, so you can continue to leverage your QuickDraw-compatible code. Also, because you receive the drawing context from a Carbon event, you do not have to worry about handling ports, creating contexts, and so on; you simply draw.

In addition, if your application used the Appearance Manager, you can now use the HITheme API which allows you to draw Aqua-compliant features using Quartz.

For more information about using Quartz, see *Quartz 2D Programming Guide*.

It’s the Future

Apple is committed to the HIViews, Carbon events, and nib files for Carbon implementations of the user interface. All new controls and other features will be based on HIView. If you want your application to take advantage of the latest features, you need to adopt the modern HIToolbox.

Adopt Incrementally

Because backwards compatibility has been a hallmark of Mac OS system software, you do not have to throw out all your older implementations to begin updating your user interface code.

- Nib files can coexist with resources in the same application. You can first adopt nibs for windows that require views or compositing, and address the rest as your schedule permits.
- An application can support both `WaitNextEvent` event dispatching and Carbon events. For example, you can convert to Carbon events first in windows that use HIView, and update the others as time permits. Another priority is to upgrade any user interface code that polls or otherwise makes poor use of CPU cycles.
- You can adopt compositing mode on a window-by-window basis.
- HIView supports both QuickDraw and Quartz, so you can migrate your drawing code incrementally as well.

Porting Steps

This chapter focuses on the steps required to modernize your user interface code, moving your application to views and composited windows. The following sections describe the porting process:

- "Convert Resources to Nibs" (page 13): Part of this process is to move away from old Dialog Manager calls.
- "Adopt Carbon Events" (page 24): If you are still using `WaitNextEvent`, now is the time to move to the more efficient Carbon event model.
- "Put Custom Content Into Views" (page 27): If you have nonstandard controls or content, you may need to implement your own views.
- "Turn On Compositing" (page 32): Only composited windows gain the full benefits of using HIViews.
- "Additional Steps" (page 32): These steps are miscellaneous changes resulting from moving to HIViews, such as translating graphics coordinates.

Before you begin adopting views, you should be familiar with the contents of *HIView Programming Guide*. This document outlines the concepts behind HIView, how the older control, windows, and menus map to the HIView world, and gives examples of implementing views. You should examine all your windows and menus to estimate the changes that would be required to adopt HIView. In most cases, the more you rely on standard user interface elements, the easier the porting process. You can choose to port on a window-by-window basis, as time and resources permit.

If you have custom controls, check to see if their functionality is now available in a standard control or view. For example, in the past combining an editable text field with an integral pop-up menu required custom code; now you can simply create an HIView-based combo box to gain the same functionality. Adopting other views, such as the scroll view, might help to reduce or eliminate large chunks of legacy code.

Convert Resources to Nibs

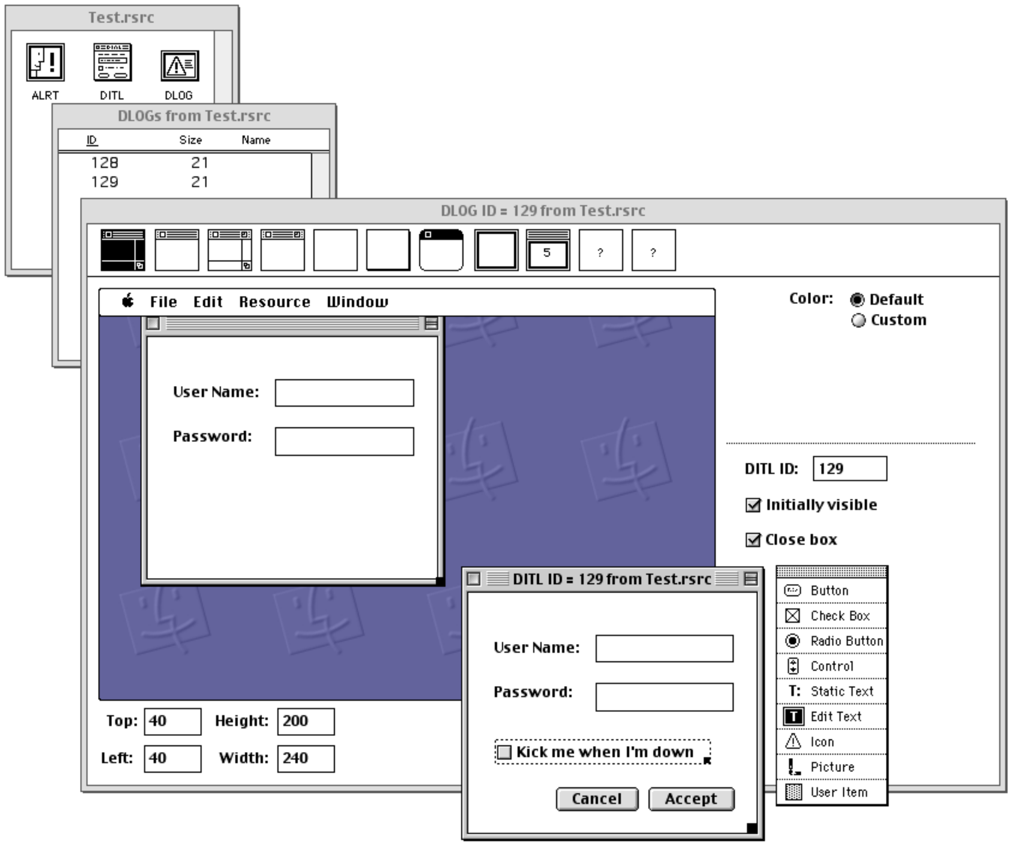
If you have legacy Mac OS code, many of your windows, controls, and menus are likely stored as resources. Similarly, you may be relying on the Dialog Manager to handle many of your windows. In both cases, in order to prepare your application for views and composited windows, you must move from resource-based interface elements to Interface Builder nibs.

Nibs provide an XML-based description for windows, controls, and menus. You create and manipulate them using the Interface Builder tool in Xcode. Interface Builder provides a simple graphical interface for building user interfaces, in which you can simply drag interface elements around to build menus, dialogs, and so on. If you are not familiar with Interface Builder, the "Designing a User Interface" section in *Xcode Quick Tour for Mac OS X* is a good introduction.

Note: You do not need to build your application with Xcode to use Interface Builder or nib files.

For Mac OS X v10.3 and later, Interface Builder (version 2.4) allows you to import older compiled resource files (`.rsrc`) files, converting them automatically into nibs. In some ways, Interface Builder is like a more modern, advanced version of the ResEdit resource editor (Figure 3-1 (page 14)).

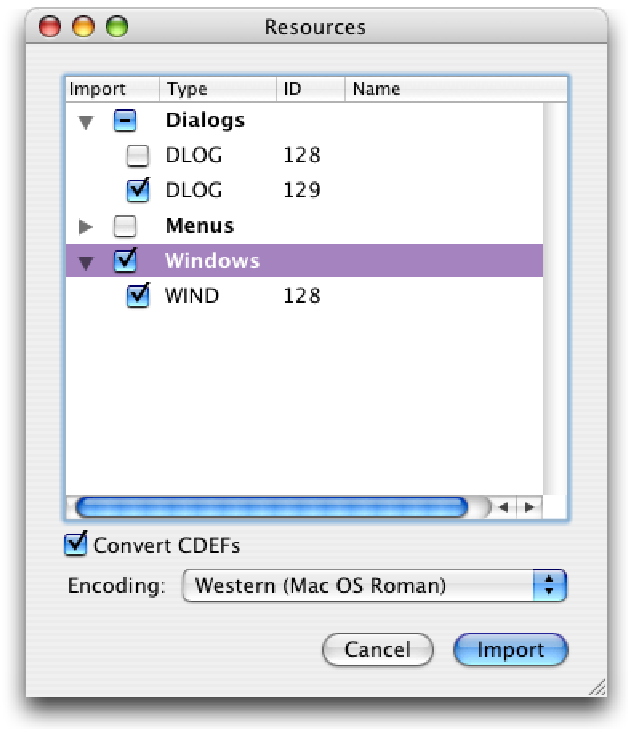
Figure 2-1 A dialog resource in ResEdit



To import a file, simply choose Import Resource File from the Import submenu of the File menu and choose your file from the resulting dialog window. Selecting a `.rsrc` file brings up a resource selection dialog as shown in Figure 3-2 (page 15).

You can choose to import some or all of the resources contained in your `.rsrc` file. You can import any of the standard `'WIND'`, `'DLOG'`, and `'MENU'` resources. Note that if your `'DLOG'` resource uses an item list (`'DITL'`) resource, Interface Builder automatically imports items in the list into the dialog window.

Figure 2-2 The resource selection dialog window



Import	Type	ID	Name
▼ ▬	**Dialogs**		
☐	DLOG	128	
✓	DLOG	129	
▶ ☐	**Menus**		
▼ ✓	**Windows**		
✓	WIND	128	

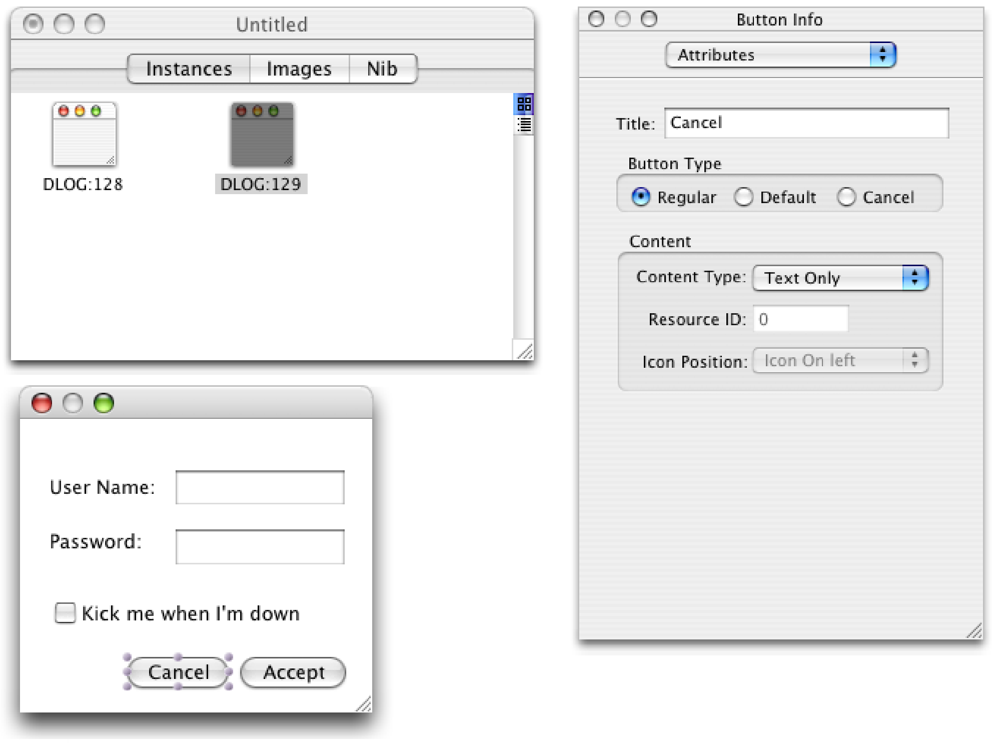
✓ Convert CDEFs

Encoding: Western (Mac OS Roman)

Note: Depending on the number of resources you are converting, you can also choose to ignore your old resources and simply rebuild your user interface elements in Interface Builder.

You should check the windows, controls, and menus to make sure that they were imported correctly. Note that in a nib file, dialog resources simply become windows, and any dialog items are translated into controls (Figure 3-3 (page 16)). Because of this mapping, you will probably need to update any Dialog Manager calls. See "Update Dialog Manager Functions" (page 18) for more information.

Figure 2-3 A dialog resource imported into Interface Builder



Title: Cancel

Button Type: [✓ Regular] Default Cancel

Content

Content Type: Text Only

Resource ID: 0

Icon Position: Icon On left

Some tips:

- The Info window in Interface Builder lets you choose Default and Cancel behavior for simple push buttons. Specifying these behaviors also enables the keyboard shortcuts (that is, Esc for Cancel, and Return or Enter for the default), so you do not need to handle them yourself.
- Be sure to specify the window class and theme brush for each window you import using the Info window. Doing so ensures that your window has the look appropriate to its type.

In addition, you should be aware of the following changes:

- If your resources used custom controls, the nib replaces them with generic custom control placeholders. While the Info window shows that you can use a procedure pointer–based custom definition, you should convert your control to an event-based HIView. Note that if you use HIView-based controls, you need to replace the custom control placeholder with the HIView placeholder.
- All user items are translated into user panes. Depending on what you used the user item for, you probably want to change this to a different control. For example, if you used it to visually group several controls, you should replace it with a group box. For simple static images and drawing, you can simply add `kEventControlDraw` handlers to the user pane that Interface Builder substituted for your item. More complex user items may need to be reimplemented as custom views. See "Put Custom Content Into Views" (page 27) for more information.
- Modern windows and controls are based on a hierarchy, which means that some controls may be embedded in others. If your old resources did not use hierarchies, you should update the resulting nib file appropriately. For example, if you used a user item to group controls, you should replace the user item with a group box and then embed the controls within the group box. In Interface Builder, dragging a control into the bounds of the group box embeds it automatically.

Note that window resources imported into Interface Builder do not automatically become Aqua compliant; if you have not already done so, you will probably have to reposition, resize, or otherwise tweak controls, text, and so on, to conform to *Apple Human Interface Guidelines*. While this may seem like an afterthought, taking the extra time to adjust your windows now pays off in a better overall user experience.

Next, you need to update your application code to load the user interface elements from the nib. Note that you need to place the nib file in the Resources folder of your application's bundle.

Listing 3-1 (page 17) shows how you would load a menu bar from a nib file.

Listing 2-1 Creating a menu bar from a nib file

```
OSStatus err;
IBNibRef theNib;

err = CreateNibReference (CFSTR("MyGuitar"), &theNib);                  // 1
if (!err)
    {
    SetMenuBarFromNib (theNib, CFSTR("GuitarMenu"));                    // 2
    DisposeNibReference (theNib);                                       // 3
    }
```

Here is what the code does:

1. The Interface Builder Services function `CreateNibReference` creates a nib reference that points to the specified file. In this case, the file is `MyGuitar.nib` (you don't need to specify the `.nib` extension when calling this function). The `CFSTR` macro converts the string into a Core Foundation string, which is the format that `CreateNibReference` expects.

2. The Interface Builder Services function `SetMenuBarFromNib` uses the nib reference to access a menu bar within the nib file. The name of the menu bar (`GuitarMenu` in this example) is the name you assigned to it in the Instances pane of the nib file window. Like the `CreateNibReference` function, `SetMenuBarFromNib` expects a Core Foundation string for the menu bar name, so it must first be converted using `CFSTR`.

   Note that `SetMenuBarFromNib` automatically sets the menu bar you specified to be visible. If for some reason you want to create a menu bar but don't want it to be immediately visible, you can call `CreateMenuBarFromNib`. You can then call `SetMenuBar` to make it the main menu bar.

   If you want to load individual menus, you can call the Interface Builder Services function `CreateMenuFromNib` after you create the nib reference.

3. When you no longer need the nib reference, you should call the Interface Builder Services function `DisposeNibReference` to remove it.

Creating a window from a nib file is similar, except that you call `CreateWindowFromNib`, as shown in Listing 3-2 (page 17).

Listing 2-2 Creating a window from a nib file

```
OSStatus err;
IBNibRef theNib;
WindowRef theWindow;

err = CreateNibReference (CFSTR("MyGuitar"), &theNib);
```

```
if (!err)
    {
    CreateWindowFromNib (theNib, CFSTR("GuitarPrefs"), &theWindow);
    if (theWindow != NULL) ShowWindow(theWindow);

    DisposeNibReference (theNib);
    }
```

The window is hidden when first created, so you need to call `ShowWindow` to make it visible. Any controls embedded in the window are automatically created and laid out at the same time.

After instantiation, you have a valid window or menu reference that your application can then use to perform other actions (obtain the menu ID, add menu items, and so on). In many cases, your existing code should just work once you've created your user interface elements from the nib file.

Update Dialog Manager Functions

Most Dialog Manager functions were based on the old resource-based method of creating windows. To ensure compatibility with composited windows, you should replace your Dialog Manager calls with more modern equivalents. If you updated dialog resources to use nibs, you must change some Dialog Manager calls. Here are a few guidelines to keep in mind:

- Calls that manipulate dialogs are typically replaced by Window Manager equivalents. Nib files do not distinguish between windows and dialogs.
- Calls that manipulate dialog items are replaced by their Control Manager or HIView equivalents.
- Event management in dialogs is now handled through Carbon events. Any event filtering callbacks must be replaced by the appropriate Carbon event handlers. See "Event Filtering" (page 21) for more infomation.
- If you have simple alert dialogs, you may use `CreateStandardAlert` rather than implement it in a nib file. See "Standard Alerts" (page 23). You can also programmatically create simple sheets for documents windows, as described in "Sheets" (page 23).

The only Dialog Manager functions still recommended are those that handle standard alerts and sheets.

Table 3-1 (page 18) lists a number of common Dialog Manager calls and their replacements.

Table 2-1 HIView-savvy Dialog Manager replacements

Dialog Manager function	HIView replacement
`AppendDITL` or `AppendDialogItemList`	`HIViewAddSubview`
`CountDITL`	Control Manager: `CountSubControls` with the root control as parent.
`FindDialogItem`	`HIViewFindByID`

Dialog Manager function	HIView replacement
`GetDialogItem`	`HIViewFindByID`. Note that when importing dialog resources into Interface Builder, dialog items are converted into controls with an ID number corresponding to the item index. For example, dialog item 1 turns into a control with control ID 1. However, the signature field is left blank, so you should set this to be your application's signature. If desired, you can also use the Control Manager function `GetIndexedSubControl`
`SetDialogItem`	Varies depending on what you want to do. • If you want to set an application-defined drawing function , you should install a drawing handler on a user pane control. See "Put Custom Content Into Views" (page 27). • If you want to set control bounds, use `HIViewSetFrame` on the view reference. •If you want to actually change the control type, you should create one of each type you want to display and enable/disable/show/hide them appropriately. For example, if you used `SetDialogItem` to change an edit text control to a static one , you should create one of each control with the same bounds. To switch, you would disable and hide one while enabling and showing the other.
`RemoveDialogItem` and `ShortenDITL`	Control Manager: `DisposeControl` or `HIViewRemoveFrom-Superview`. One advantage of using `HIViewRemoveFromSuperview` is that you still retain the view, so you can embed it elsewhere later.
`HideDialogItem` or `ShowDialogItem`	`HIViewSetVisible`
`InsertDialogItem`	`HIViewAddSubview`
`Alert`, `StandardAlert`, `StopAlert`, `NoteAlert`, `CautionAlert`, `GetAlertStage` and `ResetAlertStage`	Convert alerts to use `CreateStandardAlert` instead. See "Standard Alerts" (page 23).
`GetNewDialog`	Interface Builder Services: `CreateWindowFromNib` with appropriate nib file.
`NewFeaturesDialog`, `NewDialog`, and `NewColorDialog`	Window Manager: `CreateNewWindow`
`CloseDialog`	Not supported in Carbon. Use `DisposeWindow`.
`DisposeDialog`	Window Manager: `DisposeWindow`
`DrawDialog` and `UpdateDialog`	Draw from within `kEventControlDraw` handlers only. Do not draw from window drawing or window update handlers.
`GetDialogItemAsControl`	Control Manager: `GetIndexedSubControl` or `HIViewFindByID`.

Dialog Manager function	HIView replacement
`GetModalDialogEventMask` and `SetModalDialogEventMask`	No longer needed. Events are filtered by registering for specific Carbon events.
`ModalDialog`	Carbon Event Manager: `RunAppModalLoopForWindow` on the window you want to be application modal. See *Carbon Event Manager Programming Guide* for more details.
`New/Invoke/Dispose-UserItemUPP`	User items typically replaced by standard or custom views.
`New/Invoke/Dispose-ModalFilterUPP` and `New/Invoke/Dispose-ModalFilterYUPP`	Event filters not needed in Mac OS X. Use Carbon events instead. See "Event Filtering" (page 21) for more information.
`DialogSelect` and `IsDialogEvent`	No longer needed. Events are filtered by registering for specific Carbon events. See "Event Filtering" (page 21) for more information.
`ParamText` and `GetParamText`	Control Manager: `GetControlData` with the `kControlStaticTextCFStringTag` tag.
`GetDialogItemText` and `SetDialogItemText`	Control Manager: `GetControlData` or `SetControlData` with the `kControlEditTextCFStringTag` or `kControlStaticText-CFStringTag` tag.
`SelectDialogItemText`	Control Manager: `SetControlData` with the `kControlEditText-SelectionTag` tag.
`DialogCut`, `DialogCopy`, `DialogPaste`, and `DialogDelete`	No longer needed. The standard window event handler handles these actions for you.
`SetDialogCancelItem`, `GetDialogCancelItem`, `SetDialogDefaultItem`, and `GetDialogDefaultItem`	Carbon Event Manager: `GetWindowDefaultButton`, `SetWindowDefaultButton`, `GetWindowCancelButton`, `SetWindowCancelButton`. You can also set Default and Cancel buttons in nib file. In most cases, you should set the command ID for the button to `kHICommandOK` (Default) or `kHICommandCancel` (Cancel) so that you can handle the command event.
`GetDialogKeyboardFocusItem`	Control Manager: `GetKeyboardFocus` on the appropriate window.
`SetDialogTracksCursor`	The preferred method is to use mouse-tracking regions (as described in *Carbon Event Manager Programming Guide*) and change your cursor based on mouse-entered and mouse-exited events. However, you can also register for the `kEventMouseMoved` event and change the cursor according to its position.
`MoveDialogItem`	`HIViewMoveBy` or `HIViewPlaceInSuperviewAt`.
`SizeDialogItem`	`HIViewSetFrame`.
`AutoSizeDialog`	Control Manager: Use `GetBestControlRect` on text fields that need to be autosized.

Dialog Manager function	HIView replacement
`GetDialogTimeout` and `SetDialogTimeout`	Use Carbon event timers instead. See *Carbon Event Manager Programming Guide* for more details.
`GetDialogWindow`	Not needed in most cases, because the dialog is now a window. However, `CreateStandardSheet` creates a dialog reference,which you must convert to type `WindowRef` using this function when passing it to `ShowSheetWindow`.
`GetDialogPort`	Window Manager: `GetWindowPort`. However, you should consider using Quartz rather than QuickDraw.
`GetDialogTextEditHandle`	Control Manager: `GetControlData` with the `kControlEditText-TEHandle` on the focused edit text control. However, TextEdit is not recommended, so in most cases, you should upgrade to the Unicode edit text control.
`CreateStandardSheet`	Still supported. See "Sheets" (page 23) for more details.
`CreateStandardAlert`	Still supported. See "Standard Alerts" (page 23) for more details.

Event Filtering

If you have event filters for your dialogs, you should remove them when you convert to nibs. In most cases, any action the user takes on a control (clicking it, entering text, sliding a thumb) invokes Carbon events. Thus, registering to receive particular events and then handling them in the application essentially substitutes for dialog event filtering.

When you want to filter keyboard input to, say, a Unicode edit text field, you should install a control key filter on the control. For example, say you had event filtering–code in your dialog to allow only numbers in an edit text field, as shown in Listing 3-3 (page 21).

Listing 2-3 Filtering keyboard input from a dialog event filter

```
Boolean MyOldDialogFilter(DialogRef theDialog,
                        EventRecord *theEvent, DialogItemIndex *itemHit)
    {
    if ((theEvent->what == keyDown) || (theEvent->what == autoKey))
        {
        char c = (theEvent->message & charCodeMask);

        // return or enter key?
        if ((c == kReturnCharCode) || (c == kEnterCharCode))
            {
            *itemHit = 1;
            return true;
            }

        // tab key or arrow keys?
        if (c == kTabCharCode) return false;
        if (c == kLeftArrowCharCode) return false;
        if (c == kRightArrowCharCode) return false;
        if (c == kUpArrowCharCode) return false;
        if (c == kDownArrowCharCode) return false;

        // digits only for edittext box item #9 ?
        // pre-Carbon, this would have been:
        // ((DialogPeek)theDialog)->editField+1 == 9
        if (GetDialogKeyboardFocusItem(theDialog) == 9)
            {
            if ((c < '0') || (c > '9'))
                {
                SysBeep(1);
                return true;
                }
            }
        }
...
    return false;
    }
```

In Mac OS X (actually, Mac OS 8 and later), you can install a control key filter on your control to do the same thing, as shown in Listing 3-4 (page 22).

Listing 2-4 A control key filter

```
ControlKeyFilterResult MyEditKeyFilter(ControlRef theControl,
            SInt16 *keyCode, SInt16 *charCode, EventModifiers *modifiers)
    {
    // the edit text control can filter keys on its own
    if ((*charCode < '0') || (*charCode > '9'))
        {
        SysBeep(1);
        return kControlKeyFilterBlockKey;
        }
    return kControlKeyFilterPassKey;
    }
```

You install the key filter using the `SetControlData` function, as shown in Listing 3-5 (page 22).

Listing 2-5 Installing the control key filter

```
HIViewID hidnst = {0, 9};                                          // 1
HIViewRef numEditText;

HIViewFindByID(HIViewGetRoot(window), hidnst, &numEditText);       // 2

ControlKeyFilterUPP keyFilter =
                    NewControlKeyFilterUPP(MyEditKeyFilter);       // 3

SetControlData(numEditText, kControlEntireControl,
         kControlEditTextKeyFilterTag, sizeof(keyFilter), &keyFilter);  // 4

DisposeControlKeyFilterUPP(keyFilter);
```

Here is how the code works:

1. This example assumes that the edit text control is stored in the nib and has been assigned an HIView ID (essentially the same as a control ID).

2. To obtain a reference to the HIView in the window that contains it, you use the `HIViewFindByID` function, passing in the parent view in which you want to search (in this case, the root view).

3. The key filter is a callback function, so you should pass a universal procedure pointer (UPP) rather than a simple procedure pointer.

4. You specify the key filter for your control by calling `SetControlData` with the `kControlEditTextKeyFilterTag` tag and passing the universal procedure pointer (UPP) of your filter function as the data.

Standard Alerts

If you have simple alerts in your application, instead of converting them to nibs you can choose to use the Dialog Manager function `CreateStandardAlert` to create them on the fly. These dialogs assume only minimal user interaction (that is, Cancel and Ok). Listing 3-6 (page 23) shows how to create a simple alert.

Listing 2-6 Creating a simple alert

```
DialogRef theAlert;
DialogItemIndex itemIndex;

CreateStandardAlert(kAlertPlainAlert,                               // 1
                CFSTR("Be vewy vewy quiet."),                       // 2
                CFSTR("I'm hunting wabbits."),
                NULL, &theAlert);                                   // 3

RunStandardAlert (theAlert, NULL, &itemIndex);                      // 4
```

Here is what the code does:

1. When calling `CreateStandardAlert`, passing `kAlertPlainAlert` specifies that you want the application icon to be used. Other possible constants you can pass are `kAlertNoteAlert`, `kAlertCautionAlert`, and `kAlertStopAlert`. Note that in most cases you should simply use the plain alert. See *Apple Human Interface Guidelines* for more information. Your application icon is automatically added to an alert icon in accordance with the Apple guidelines.

2. The Core Foundation strings (created using `CFSTR`) specify the alert message you want displayed. The second string contains the smaller, informative text.

3. If you have a custom parameter block describing how to create the alert, you would pass it here. Otherwise pass `NULL`. On return, `theAlert` contains a reference to the new alert.

4. `RunStandardAlert` displays the alert and puts the window in an application-modal state. When the user exits the alert (by clicking OK or Cancel), `itemIndex` contains the index of the control the user clicked.

Sheets

Sometimes you may want to make your alerts document-modal, in which case you should implement them as sheets.

If you want to create a simple alert that appears as a sheet, you can call the function `CreateStandardSheet`. This function is analogous in format to the `CreateStandardAlert` function. However, it includes an additional parameter to specify an event target. When the user dismisses the sheet alert (by clicking OK or Cancel), the system sends a command event (class `kEventClassCommand`, type `kEventCommandProcess`) to the specified event target. You can use this event to determine which control the user clicked.

In most cases, the event target you specify in `CreateStandardSheet` should be the sheet's parent window. For example, you might want to do the following when you want to display a sheet:

1. Install an event handler on the parent window for the `kEventCommandProcess` event.
2. Call `CreateStandardSheet` and `ShowSheetWindow` to create and display the sheet. Note that `CreateStandardSheet` creates a dialog reference, and `ShowSheetWindow` requires a window reference. To convert the reference appropriately, you need to use the `GetDialogWindow` function as follows:

   ```
   DialogRef theSheet;
   CreateStandardSheet (…, &theSheet);
   ShowSheetWindow (GetDialogWindow (theSheet));
   ```

3. When your parent window receives the `kEventCommandProcess` event, process it accordingly, based on the command ID.
4. Remove the event handler from the parent window.

By installing the `kEventCommandProcess` event handler only when the sheet is visible, you ensure that the handler receives events only from the sheet, not from any other controls.

Adopt Carbon Events

In order to adopt HIViews, your windows must use Carbon events for event processing. You can think of Carbon events as the methods that complement the HIView data store objects. You register for only the Carbon events that you are interested in and implement the event handlers as callback functions.

Because Mac OS X supports both Carbon events and the `WaitNextEvent` event model, you can choose to adopt Carbon events at first only for those windows that will use HIViews. If you are converting Dialog Manager dialog windows, most of the event filtering done in event filters and modal dialogs are replaced by Carbon events.

If you do not plan to implement Carbon events or views immediately throughout your application, you should still consider targeted adoption to improve performance. For example, using Carbon events to eliminate old system-polling code (such as calls to `StillDown`) can significantly improve your application's responsiveness.

For more details about the Carbon Event Manager, read *Carbon Event Manager Programming Guide*. For an example of how to move from the `WaitNextEvent` event model to Carbon events, see *Carbon Porting Guide*.

Standard Handlers

The major benefit for Carbon events is the use of standard handlers. That is, the Carbon Event Manager has implemented handlers for common events. For windows and controls, these handlers are installed when you install the standard window handler. For example, the standard window handler includes a routine to drag windows. As a result, your windows can be dragged without your having to write any code. Of course, if you wanted some special dragging behavior for your window, you can override the standard handler with one of your own.

Standard handlers for windows and controls are installed when you set the `kWindowStandardHandlerAttribute` attribute for the window. You can set this from the Info window in Interface Builder or programmatically when you call the Window Manager function `CreateNewWindow`. Note that because this attribute is set on a window-by-window basis, you can choose to use the standard handler in some windows but not others.

Menus and the application itself have the equivalent of standard handlers; these handlers are installed when you call the Carbon Event Manager function `RunApplicationEventLoop`.

Because the standard handlers do so much, you should check your event handling code against the standard hander behavior for those particular events; you may discover that you no longer need to implement certain actions. See information on specific events in *Carbon Event Manager Reference* to determine their standard handler behavior.

A good rule of thumb for determining standard handler behavior is to test your user interface elements without installing any additional event handlers. Launching the Carbon Simulator application, available from the Test Interface menu item in the Interface Builder File menu, is a simple way to do this. Whatever behavior is present (tracking, keyboard focus, dragging, and so on) is provided by standard event handlers.

Command Events

To handle the behavior of most simple single-action controls, you should use command events. Doing so requires you to assign a unique command ID to each control, typically from the Interface Builder Info window. When the user activates your control, the Carbon Event Manager sends a `kEventCommandProcess` event containing the control's command ID to your application. One major advantage of the command event is that you can assign the same command ID to both a menu and a control; you can then handle both cases with a single event handler.

The Carbon Event Manager defines command IDs for many common commands, such as OK, Cancel, Cut, and Paste. You can also define your own for application-specific commands. Your event handler for the `kEventCommandProcess` event can then determine which command ID was sent and take appropriate action.

Important: Command IDs containing all lower case letters are defined by Apple; if you create nonstandard command IDs, they must contain at least one upper case letter.

You assign the command ID to a menu item in the Attributes pane of the Interface Builder Info window. You can also call the Menu Manager function `SetMenuItemCommandID`.

The `kEventCommandProcess` event indicates that your menu item was selected. The actual command ID is stored within an `HICommandExtended` structure in the event reference. You must call the Carbon Event Manager function `GetEventParameter` to retrieve it, as shown in Listing 3-7 (page 26).

Note: The `HICommandExtended` structure supersedes the `HICommand` structure. The extended version adds fields for a control or window reference, while the original `HICommand` structure had space only for menu information.

Listing 2-7 Obtaining the command ID from the event reference

```
HICommandExtended commandStruct;
UInt32 the CommandID;

GetEventParameter (event, kEventParamDirectObject,                    // 1
                    typeHICommand, NULL, sizeof(commandStruct),
                    NULL, &commandStruct);

theCommandID = commandStruct.commandID;                               // 2
```

Here is what the code does:

1. When calling `GetEventParameter`, you must specify which parameter you want to obtain. For command events, the direct object (`kEventParamDirectObject`) is the `HICommandExtended` structure, which describes the command that occurred.

2. The command ID of the control (or menu) that generated the event is stored in the `commandID` field of the `HICommandExtended` structure.

To respond to events from menus, you should install your command event handler at the window or application level. Doing so also allows you to use the same handler to catch command events coming from controls, if so desired. Also, attaching your handler at the window level makes sense if you have menu items that apply to one type of document window but not to another, because command events are dispatched only to the active window, not to any other window.

After handling a command, your application may need to change the state of a menu item. For example, after saving a document, the Save menu item should be disabled until the document changes. Whenever the status of a command item might be in question, the system makes a note of it. When the user takes an action that may require updating the status (such as pulling down a menu), your application receives a `kEventCommandUpdateStatus` event. To make sure that the states of your menus are properly synchronized, you should install a handler for the `kEventCommandUpdateStatus` event. This handler should check the attributes bit of the command event to determine which items may need updating. Some examples of possible updates include:

- Enabling or disabling menu items
- Changing the text of a menu item (for example, from Show *xxxx* to Hide *xxxx*).

If the `kHICommandFromMenu` bit in the `attributes` field of the `HICommandExtended` structure (shown in Listing 3-8 (page 26)) is set, then you should check the menu item in question to see if you need to update it.

Listing 2-8 The extended HICommand structure

```
struct HICommandExtended {
   UInt32 attributes
   UInt32 commandID
   union     {
        controlRef;
        windowRef;
        struct         {
            MenuRef menuRef;
            MenuItemIndex menuItemIndex;
        } menu;
    } source;
};
typedef struct HICommandExtended HICommandExtended;
```

Put Custom Content Into Views

The transition to views and composited windows is simpler if you use only the standard Apple-supplied user interface elements. However, in many cases you may have custom content that has no standard equivalent.

One constraint of the compositing drawing model is that all your drawing must occur in a view. In the past, if you wanted to modify a window, you could simply draw to the screen as necessary. The drawback was that drawing was not predictable and could result in unnecessary redraws. For example, you may have used a standard dialog and then added additional images by drawing directly onscreen. This method no longer works in compositing mode, because you can no longer dictate when you can draw to the screen.

If you have custom content, you must draw it in a manner that supports compositing mode. This means that you can no longer draw:

- By calling `DrawControl`, `Draw1Control`, or `UpdateControls`
- During a classic update event or `kEventWindowUpdate` event.
- During a `kEventWindowDrawContent` event

Typically this means you must create a view that handles the `kEventControlDraw` event to draw its content. In addition, you should be aware of the following restrictions:

- Don’t erase behind your view when before drawing.
- Make sure you draw using view-local coordinates (rather than, say global coordinates, or content region–local coordinates).
- If you draw with Quartz, you must draw into the Core Graphics context supplied to you in the draw event.

If you want to update your view content, you must first mark the areas to be redrawn. You do so by calling the `HIViewSetNeedsDisplay` or `HIViewSetNeedsDisplayInRegion` function, and then draw only when the view receives the `kEventControlDraw` event.

Note: While Apple recommends implementing custom controls as HIView subclasses (using `HIObjectRegisterSubclass`), it is possible for older CDEFs to work in compositing mode if they adhere to the above restrictions. For example, if you want to draw using Quartz, a custom control can install a `kEventControlDraw` handler on its control reference and use that single event to draw, while otherwise supporting the classic CDEF interface.

Custom Drawing in User Panes

If you have only simple drawing needs, you can attach a `kEventControlDraw` handler to a user pane control. Conveniently, Interface Builder converts dialog user items to user panes when importing from resources.

Listing 3-9 (page 28) shows how you would install a simple, one-event drawing handler onto a user pane.

Listing 2-9 Installing a draw ing handler onto a user pane

```
WindowRef window;

HIViewRef       userPane;
static const HIViewID userPaneID = { 'Moof', 127 };                    // 1

static const EventTypeSpec myEventTypes =                              // 2
                        { kEventClassControl, kEventControlDraw };

HIViewFindByID (HIViewGetRoot(window), userPaneID, &userPane);         // 3
InstallControlEventHandler (userPane, myUserDraw, 1, &myEventTypes,    // 4
                            userPane, NULL);
```

Here is what the code does:

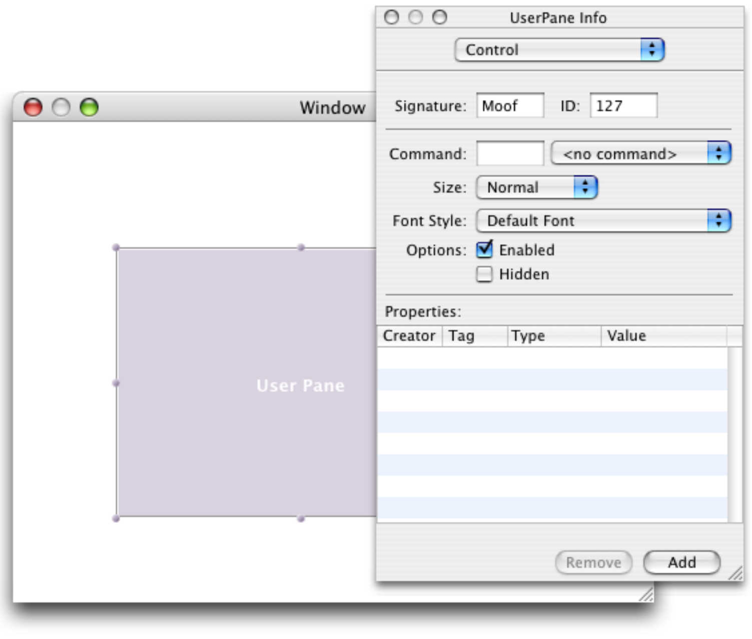
1. Identifies a view generated from a nib by specifying its HIView ID, which is identical to a control ID. This must match the application signature and the view/control ID value you set for the user pane in the Info window, as shown in Figure 3-4 (page 29).

2. Specify the events you want to register for by inserting them in an array of type `EventTypeSpec`. Each event is defined by its class and kind. In this case, you are only registering for the one control class event, `kEventControlDraw`.

3. Obtains the user pane's HIView reference. This example specifies the root view (obtained using the `HIViewGetRoot` function) as the parent within which you want to look for the user pane.

4. Calls the Carbon Event Manager macro `InstallControEventHandler` (a macro variant of `InstallEventHandler`) to register `myUserDraw` as the handler for your user pane drawing event. This example also passes a reference to the user pane in the application-defined user data parameter. Doing so eliminates the need to call `GetEventParameter` in the event handler to obtain the user pane reference.

Figure 2-4 Specifying the user pane signature and HIView ID in the Info window



Listing 3-9 (page 28) shows a possible implementation for the `myUserDraw` function. This example draws a 10-unit wide border inside the user pane's bounds.

Listing 2-10 A simple drawing handler for the user pane

```
pascal OSStatus myUserDraw (EventHandlerCallRef nextHandler,
                                EventRef theEvent, void * userData)
{
    OSStatus err;

    CGContextRef theCGContext;
    HIRect paneBounds, myHIRect;

    HIViewRef thePane = (HIViewRef)userData;                        // 1

    err = GetEventParameter( theEvent, kEventParamCGContextRef,     // 2
                        typeCGContextRef, NULL, sizeof( CGContextRef ),
                        NULL, &theCGContext );

    HIViewGetBounds (thePane, &paneBounds);                         // 3

    myHIRect = CGRectInset (paneBounds, 5.0, 5.0);                  // 4

    CGContextStrokeRectWithWidth (theCGContext, myHIRect,10.0);     // 5

    return err;
}
```

Here is what the code does:

1. Casts the incoming user data (which is a reference to the user pane) to be type `HIViewRef`.

2. Calls the Carbon Event Manager function `GetEventParameter` to obtain the Core Graphics drawing context for the user pane. You need to specify this context whenever you make any Quartz drawing calls.

3. Obtain the bounds of the user pane by calling `HIViewGetBounds`. Note that these bounds are of type `HIRect`, which is structured differently from the old QuickDraw `Rect` type.

4. Lines drawn in Quartz are centered on the line you specify; that is, a line of width 10.0 units extends 5.0 units to either side of the specified drawing line. To keep the line from extending beyond the bounds of the user pane, the code shrinks the drawing rectangle by 5.0 units on each side. The drawing context is clipped to the bounds of the view when you receive it, so anything drawn outside the bounds will not appear. While you could adjust each of the `HIRect` fields individually, the simplest way is to use the `CGRectInset` function.

5. `CGContextStrokeRectWithWidth` draws a rectangle with a line of the specified width (10.0 units in this example).

If you have nonstandard controls or other widgets in your window, you need to implement them as custom HIViews. You can implement a custom view as a subclass of a standard control or as a subclass of the base HIView class.

Note: If you are using C++, you may want to look at the HIFramework sample code installed in Developer/Examples/Carbon/HIFramework provides simple wrapper classes for many HIView APIs as well as an HIView subclass to use as your base class when using C++.

Subclassing Standard Controls

Because of the object-oriented nature of views, subclassing existing views often reduces the amount of code you need to write. For example, if you have a custom round pushbutton, you can subclass the standard pushbutton view and override only the drawing, hit-testing, and region calculation event handlers. Other functionality, such as mouse tracking, is inherited from the pushbutton view.

However, if you subclass an existing control, there is currently no simple way to set or change its attributes from the nib file. For example, say you create a custom text holder `kHIViewMyWhizzyTextClassID` that is a subclass of the standard static text control (`kHIStaticTextViewClassID`). If you specify `kHIViewMyWhizzyTextClassID` directly in the HIView custom element in Interface Builder, you cannot easily set the base class attributes such as the text size or font. You would have to call `SetControlFontStyle` programmatically within your application.

The alternative is to attach the event handlers that make up your subclass to the instance of the standard control.

For example, for the static text case, you would create a standard text control in the nib file, which lets you set the text, font, size, and so on. After calling `GetWindowFromNib`, you need to obtain the HIView reference for the standard text control, (for example, by using `HIViewFindByID`) then call `InstallControlEventHandler` to install your custom event handlers onto it. These handlers override any existing handlers (that is, your `kEventControlDraw` handler overrides the standard one defined for the control). The net result is a control that behaves as your subclass.

If you choose to create a true subclass of an HIView element, you must write construct and destruct handlers for your view in addition to your usual event handlers. You register your subclass using `HIObjectRegisterSubclass`. For more information about creating view subclasses, see "A Porting Example: Converting a User Item to a Custom View" (page 33) and "Creating Custom Views" in Introducing HIView.

Custom Views in Nib Files

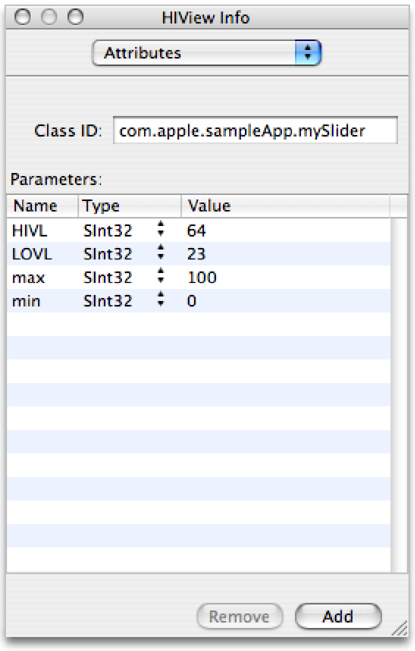
You'll find when you use nib files in your application that some of the procedures for creating custom views are slightly different than if you were creating them programmatically. For example, you would not use `HIObjectCreate` to instantiate your view, because that is done for you when you call `CreateWindowFromNib`.

You should use the HIView element in Interface Builder as the placeholder for your custom view rather than the custom control element. One major reason for this is that the HIView element allows you to specify any number of parameters comparable to the ones you would specify in your `kEventHIObjectInitialize` event. These parameters could correspond to initial state, color, title text, and so on. Note that you should set the view's bounds from the Size tab of the Info window, not as an input parameter.

Important: If you use the nib file to hold your custom HIView, you must specify any initialization parameters in the nib using the Info window. Due to the nature of creating an HIView from a nib, your application never gets the opportunity to create an initialization event for the view before the view is instantiated.

You specify your input or initialization parameters in the Interface Builder Info window under Attributes, as shown in Figure 3-5 (page 31).

Figure 2-5 Specifying input parameters in the Info window



Class ID: com.apple.sampleApp.mySlider

Parameters:

Name	Type	Value
HIVL	SInt32	64
LOVL	SInt32	23
max	SInt32	100
min	SInt32	0

The ClassID is the actual string representing your custom view class. To ensure uniqueness, you should specify this ID in the form *CompanyName.Application.ClassName*

You add parameters by hitting the Add button and then specifying the parameter name, type, and value. The parameter name should be a unique four-character string. Where possible, you should use standard Apple-defined control tags, such as the control collection tag constants, to set common values such as control value, bounds, and so on. You cannot set more complex data, such as pointers to structures, window references, and so on, from the Info window.

Turn On Compositing

An application window does not use the composited drawing model (and therefore can't gain the full benefits of HIViews) unless you specify the `kWindowCompositingAttribute` attribute. You can set this from the Info window in Interface Builder, or programmatically when you create the window using `CreateNewWindow`. Note that you cannot change this attribute after instantiating the window.

Composited windows keep track of the layering hierarchy of their views, drawing only when necessary and drawing only the visible portions of each view.

Additional Steps

Here are some additional things to keep in mind during the porting process:

- Any code that moves or positions items within a window may need to be updated to reflect the new view-relative coordinate system (that is, the local or frame coordinates).
- When importing dialogs into Interface Builder, items in a dialog list are automatically assigned an HIView ID (control ID) equivalent to its item index. That is, the first item in a dialog is given an HIView ID of {0,1}, the second {0.2}, and so on. This assignment may make it easier to update code for specific dialog items. After importing, you should set the signature field for each item to be your application's signature.
- You may want to consider using some of the `HILayout` functions to automatically position views with respect to the parent window or each other.
- If you need to add custom data to a view, you can use the Info window in Interface Builder to add properties or the `GetControlProperty` and `SetControlProperty` functions .

A Porting Example: Converting a User Item to a Custom View

This chapter describes how you might convert a dialog-based user item into a custom HIView.

This example user item/view draws a black box outline, sized to fit inside its bounds. Within the box is a square spot that the user can move or drag around by clicking in the box. The work needed to convert the user item covers many of the steps outined in "Porting Steps" (page 13): updating Dialog Manager functions, adopting Carbon events, adopting Quartz, and creating a custom view.

- "The Old Dialog Manager Code" (page 33) summarizes the Dialog Manager code used to implement the user item and describes how you might update it to use views.
- "The New Custom View" (page 36) describes a possible implementation for the new custom view, based on the recommendations given in the previous section.

The Old Dialog Manager Code

This section reviews the Dialog Manager code used to create the custom item, along with the recommendations for how to update it. This example assumes a worst-case scenario of System 6 or System 7–era code.

Listing 4-1 (page 33) shows a section of the dialog creation code dealing with the user item.

Listing 3-1 Creating the dialog with the user item

```
void ThisOldDialog(void)
    {
    SInt16 itemHit;
    DialogItemType itemType;
    Handle itemHandle;
    Rect itemBox;

    DialogRef theDialog = GetNewDialog(256, NULL, (WindowRef)-1L);          // 1
    if (theDialog == NULL) return;
    ...
    // Setting the draw proc for the user item
    GetDialogItem(theDialog, 13, &itemType, &itemHandle, &itemBox);        // 2
    gUserH = (itemBox.left + itemBox.right) / 2;
    gUserV = (itemBox.top + itemBox.bottom) / 2;
    SetDialogItem(theDialog, 13, itemType, (Handle)&MyOldDrawUserItem,     // 3
                  &itemBox);
    ...
    DisposeDialog (theDialog);                                              // 4
    }
```

To update this code, you need to make the following changes:

1. Replace the dialog resource (ID 256) with a nib file–based window.

2. Instead of calling `GetDialogItem` to obtain the user item's bounds, call `HIViewGetBounds`, specifying the HIView reference of the custom view.
3. Instead of using `SetDialogItem` to set a draw handler for the user item, register your view for the `kEventControlDraw` event and do your drawing from the handler you specify for that event.
4. Call `DisposeWindow` to remove the dialog because you have made it a window instead.

The old drawing function simply draws the box and the current position of the spot using QuickDraw calls, as shown in Listing 4-2 (page 34).

Listing 3-2 The user item drawing function

```
void MyDrawUserItem(DialogRef theDialog, DialogItemIndex itemNo)              // 1
    {
    DialogItemType itemType;
    Handle itemHandle;
    Rect itemBox;
    GetDialogItem(theDialog, itemNo, &itemType, &itemHandle, &itemBox);       // 2

    CGrafPtr savePort;                                                        // 3
    GetPort(&savePort);
    SetPortDialogPort(theDialog);

    PenState penState;                                                        // 4
    GetPenState(&penState);

    PenSize(3, 3);
    if (itemType & itemDisable)
        {
        Pattern gray;
        PenPat(GetQDGlobalsGray(&gray));
        }
    FrameRect(&itemBox);
    Rect userRect = {gUserV-4, gUserH-4, gUserV+4, gUserH+4};
    PaintRect(&userRect);

    SetPenState(&penState);
    SetPort(savePort);
    }
```

To update this code, you want to make the following changes:

1. Incorporate the drawing function into a `kEventControlDraw` handler for the custom view.
2. Instead of calling `GetDialogItem` to obtain information about the user item, obtain this information directly from the draw event using the Carbon Event Manager `GetEventParameter` function.
3. Update all QuickDraw calls to use Quartz. Instead of worrying about graphics ports, you can obtain the Quartz drawing context for your custom view from the drawing event using the `GetEventParameter` function.
4. Replace the QuickDraw drawing primitive functions to draw the item's bounding box and spot with their Quartz equivalents (for example, `CGContextStrokeRect` replaces `FrameRect`)

Mouse clicks are handled within the dialog's event filter, as shown in Listing 4-3 (page 35).

Listing 3-3 Dialog event filter to process user item clicks

```
Boolean MySystem6or7DialogFilter(DialogRef theDialog,                    // 1
                        EventRecord *theEvent, DialogItemIndex *itemHit)
    {
    …
    // we got a click!
    if (theEvent->what == mouseDown)                                     // 2
        {
        DialogItemType itemType;
        Handle itemHandle;
        Rect itemBox;
        GetDialogItem(theDialog, 13, &itemType, &itemHandle, &itemBox);  // 3

        CGrafPtr savePort;
        GetPort(&savePort);
        SetPortDialogPort(theDialog);
        Point thePoint = theEvent->where;                                // 4
        GlobalToLocal(&thePoint);
        Boolean inside = PtInRect(thePoint, &itemBox);                   // 5

        // is the click inside the user item?
        if (inside)
            {
            // let's constrain and move the spot!
            // it's possible to track the spot here but it's complex
            // so we just move on the click and don't track.
            // that's typical of dialog's user items of that era.
            Rect userRect1 = {gUserV-4, gUserH-4, gUserV+4, gUserH+4};
            EraseRect(&userRect1);                                       // 6
            InvalWindowRect(GetDialogWindow(theDialog), &userRect1);
            gUserH = thePoint.h;
            gUserV = thePoint.v;
            if (gUserH < itemBox.left+4) gUserH = itemBox.left+4;
            if (gUserH > itemBox.right-4) gUserH = itemBox.right-4;
            if (gUserV < itemBox.top+4) gUserV = itemBox.top+4;
            if (gUserV > itemBox.bottom-4) gUserV = itemBox.bottom-4;
            Rect userRect2 = {gUserV-4, gUserH-4, gUserV+4, gUserH+4};
            InvalWindowRect(GetDialogWindow(theDialog), &userRect2);
            }
        SetPort(savePort);
        }

    return false;
    }
```

To update this code, you need to make the following changes:

1. Handle mouse clicks in a Carbon event handler rather than an event filter.

2. Replace the mouse-down event handler with Carbon event handlers for the `kEventControlHitTest` and `kEventControlTrack` events. The hit test event requires you to tell the system what part of the view the user clicked, while the track event is sent if the user moves the mouse while it is down.

3. Call `GetEventParameter` to obtain the view reference, bounds, and the Quartz drawing context from the Carbon event instead of calling `GetDialogItem`.
4. You will have to change the code to translate the mouse coordinates. When converting to use views, the mouse position you receive from the Carbon event is automatically converted to be in the local coordinates of the view.
5. You do not have to compare the mouse position to the bounds. Your view receives a Carbon event only if the mouse click occurs in the view.
6. Call `HIViewSetNeedsDisplay` or `HIViewSetNeedsDisplayInRegion` to mark a view or region as needing to be redrawn. You no longer need to call `EraseRect` or `InvalRect`.

The New Custom View

To reproduce the user item in the HIView world, you implement it as a custom view. A custom view is comprised of a class name identifier, a data structure to hold the instance data, and a collection of Carbon event handlers. Listing 4-4 (page 36) shows an example class name and structure

Listing 3-4 Defining a custom view class and instance structure

```
#define kCustomSpotViewClassID                                               // 1
                        CFSTR("com.apple.sample.dts.HICustomSpotView")

typedef struct                                                               // 2
    {
    HIViewRef   view;
    HIPoint     spot;
    }
CustomSpotViewData;
```

1. Select a unique class name for your view. You pass this name to the `HIObjectRegisterSubclass` function, and you specify it in your nib file for the custom HIView element.
2. Hold the view's instance data in a `CustomSpotViewData` structure. At the bare minimum, it must contain the HIView reference for your view. In this example, the instance data also includes the coordinates of the spot inside the view.

Before you instantiate a window containing the custom view (whether from a nib file or by calling `HIObjectCreate`), you need to register your subclass by calling the `HIObjectRegisterSubclass` function, as shown in Listing 4-5 (page 36).

Listing 3-5 Registering a custom view

```
HIObjectClassRef theClass;
EventTypeSpec kFactoryEvents[] =                                             // 1
        {
            { kEventClassHIObject, kEventHIObjectConstruct },
            { kEventClassHIObject, kEventHIObjectInitialize },
            { kEventClassHIObject, kEventHIObjectDestruct },
            { kEventClassControl, kEventControlHitTest },
            { kEventClassControl, kEventControlTrack },
            { kEventClassControl, kEventControlDraw }
        };
HIObjectRegisterSubclass(kCustomSpotViewClassID, kHIViewClassID,          // 2
        0, CustomSpotViewHandler, GetEventTypeCount(kFactoryEvents),
        kFactoryEvents, 0, &theClass);
```

Note: If you are subclassing an existing view, you need to specify its class ID in place of the base HIView class ID in this example. The header files `ControlDefinitions.h`, `HIView.h`, and `MacTextEditor.h` define the class IDs for all standard views and controls.

1. Pass an `EventTypeSpec` array containing the events that you want your view to handle.

   All custom views must handle the `kEventHIObjectConstruct` and `kEventHIObjectDestruct` events. The `kEventHIObjectInitialize` event lets you perform any needed initializations.

   The control event class contains the events that describe the unique behavior of your view. Just about any custom view requires a `kEventControlDraw` handler to draw your content, and a `kEventControlHitTest` handler to provide feedback to the system about what part of the view the user clicked. This example also includes the `kEventControlTrack` event handler to track the mouse while it is down within the view.

2. Register your view using `HIObjectRegisterSubclass`, specifying the callback function to handle all your view's events. This handler is just like any other Carbon event handler, except that the instance data structure for your view is passed to you in the user data (`inRefCon`) parameter.

Listing 4-6 (page 37) shows a possible implementation for your custom view's event handler.

Listing 3-6 An event handler for the converted user item

```
pascal OSStatus CustomSpotViewHandler(EventHandlerCallRef inCaller,
                                      EventRef inEvent, void* inRefcon)
    {
    OSStatus result = eventNotHandledErr;
    CustomSpotViewData* myData = (CustomSpotViewData*)inRefcon;

    switch (GetEventClass(inEvent))                                   // 1
        {
        case kEventClassHIObject:                                     // 2
            switch (GetEventKind(inEvent))
                {
                case kEventHIObjectConstruct:
                    {
                    myData = (CustomSpotViewData*)
                                calloc(1, sizeof(CustomSpotViewData));
                    GetEventParameter(inEvent,kEventParamHIObjectInstance,
                         typeHIObjectRef, NULL, sizeof(myData->view), NULL,
                         &myData->view);
                    result = SetEventParameter(inEvent, kEventParamHIObjectInstance,
                                     typeVoidPtr, sizeof(myData), &myData);
                    break;
                    }
                case kEventHIObjectInitialize:
                    {
                    HIRect bounds;
                    GetEventParameter(inEvent, kEventParamBounds, typeHIRect, NULL,
                                         sizeof(bounds), NULL, &bounds);
                    myData->spot.x = CGRectGetMidX(bounds) - CGRectGetMinX(bounds);
                    myData->spot.y = CGRectGetMidY(bounds) - CGRectGetMinY(bounds);

                    HIViewSetVisible(myData->view, true);
                    break;
                    }

                case kEventHIObjectDestruct:
                    {
                    free(myData);
                    result = noErr;
                    break;
                    }

                default:
                    break;
                }
            break;

        case kEventClassControl:
            switch (GetEventKind(inEvent))
                {
                case kEventControlDraw:                                          // 3
                    {
                    CGContextRef    context;
                    HIRect          bounds;
                    result = GetEventParameter(inEvent,                          // 4
                             kEventParamCGContextRef, typeCGContextRef, NULL,
                                                 sizeof(context), NULL, &context);
                    HIViewGetBounds(myData->view, &bounds);                      // 5

                    CGContextSetRGBStrokeColor(context, 0.0, 0.0, 0.0, 0.7);     // 6
                    CGContextSetRGBFillColor(context, 0.0, 0.0, 0.0, 0.7);

                    CGContextSetLineWidth(context, 3.0);                         // 7
                    CGContextStrokeRect(context, bounds);

                    HIRect spot = { {myData->spot.x - 4.0, myData->spot.y - 4.0},  // 8
                                     {8.0, 8.0} };
                    CGContextFillRect(context, spot);

                    result = noErr;
                    break;
                    }

                case kEventControlHitTest:                                       // 9
                    {
                    HIPoint pt;
                    HIRect  bounds;
                    GetEventParameter(inEvent, kEventParamMouseLocation,         // 10
                                        typeHIPoint, NULL, sizeof(pt), NULL,&pt);
                    HIViewGetBounds(myData->view, &bounds);
                    ControlPartCode part = (CGRectContainsPoint(bounds, pt))           // 11
                                            ?kControlButtonPart:kControlNoPart;
                    result = SetEventParameter(inEvent, kEventParamControlPart,        // 12
                                     typeControlPartCode, sizeof(part), &part);
                    break;
                    }

                case kEventControlTrack:                                               // 13
                    {
                    MouseTrackingResult mouseStatus;
                    ControlPartCode partCode;
                    Point theQDPoint;
                    Rect windBounds;

                    HIPoint theHIPoint;
                    HIRect  bounds;

                    HIViewGetBounds(myData->view, &bounds);

                    GetWindowBounds (GetControlOwner(myData->view),                    // 14
                                         kWindowStructureRgn, &windBounds);

                    GetEventParameter(inEvent, kEventParamMouseLocation,               // 15
                        typeHIPoint, NULL, sizeof(theHIPoint), NULL, &theHIPoint);

                    mouseStatus = kMouseTrackingMouseDown;
                    while (mouseStatus != kMouseTrackingMouseUp)
                        {
                        partCode = (CGRectContainsPoint(bounds, theHIPoint))
                                    ?kControlButtonPart:kControlNoPart;

                        if (partCode == kControlButtonPart)                            // 16
                            {
                            if (theHIPoint.x < bounds.origin.x+4)                      // 17
                                                theHIPoint.x = bounds.origin.x + 4;
                            if (theHIPoint.x > bounds.origin.x + bounds.size.width-4)
                                theHIPoint.x = bounds.origin.x + bounds.size.width-4;
                            if (theHIPoint.y < bounds.origin.y+4)
                                                theHIPoint.y = bounds.origin.y + 4;
                            if (theHIPoint.y > bounds.origin.y + bounds.size.height-4)
                                theHIPoint.y = bounds.origin.y+bounds.size.height-4;

                            myData->spot = theHIPoint;
                            HIViewSetNeedsDisplay(myData->view, true);                 // 18
                            }

                        TrackMouseLocation ((GrafPtr)-1,                               // 19
                                                &theQDPoint, &mouseStatus);

                        theHIPoint.x = theQDPoint.h - windBounds.left;                 // 20
                        theHIPoint.y = theQDPoint.v - windBounds.top;

                        HIViewConvertPoint(&theHIPoint, NULL, myData->view);           // 21

                        }
                    break;
                    }
```

```
                default:
                    break;
                }
            break;

        default:
            break;
        }

    return result;
}
```

Here is how the code works:

1. When the view receives an event, obtain the event class by calling `GetEventClass`.
2. Implement the HIObject class handlers. The HIObject event class is made up of events that concern the creation and destruction of the HIObjects (of which HIView is a subclass). The actions you take here are essentially the same for any custom view:
   - `kEventHIObjectConstruct` requires you to allocate memory for your view's instance data, and then set a pointer to this data in the construct event by calling `SetEventParameter`. The structure you allocate here is then passed to your event handler when subsequent events occur.
   - `kEventHIObjectInitialize` gives you an opportunity to perform any initialization. Typically, you use this event to process any input parameters passed to your view. If you are using nibs, you can extract any parameters you specified in the Attributes pane of Interface Builder's Info window.
   - `kEventHIObjectDestruct` requires you to free the memory you allocated for your view.

   For more details about these events, see *HIView Programming Guide*.
3. Implement a drawing handler. The system sends a `kEventControlDraw` event to your view whenever it needs to be redrawn, either due to an external change or because your application called `HIViewSetNeedsDisplay` for the view. As emphasized before, all your drawing must take place within this handler.
4. For the `kEventControlDraw` event, obtain the Quartz drawing context for the view by calling `GetEventParameter` with the `kEventParamCGContextRef` parameter tag. This context is automatically clipped to only the portion that needs to be drawn.
5. Obtain the bounds of the view by calling `HIViewGetBounds`. Note that these bounds are in a structure of type `HIRect`, not type `Rect`.
6. Set the Quartz stroke and fill color. These calls are analogous to setting the pen and fill pattern in QuickDraw.
7. Set the stroke width and draw the box outline. `CGContextStrokeRect` is the Quartz equivalent of the QuickDraw `FrameRect` function.
8. Draw the 8-by-8 spot centered at the position stored in the view's instance data. The `CGContextFillRect` function is analogous to the QuickDraw `PaintRect` function.

9. Implement a hit test handler. The system sends a `kEventControlHitTest` event to your view whenever the user clicks in it. Your handler must determine what part of the view was clicked and report that to the system. Complex controls with multiple parts (such as a scroll bar, which has a track area, a thumb, and increment/decrement buttons) may require different responses depending on which part was hit.

10. Obtain the mouse position from the `kEventControlHitTest` event using `GetEventParameter`.

11. Determine if the mouse down happened within the view using the Quartz function `CGRectContainsPoint` function. If so, set the part code to `kControlButtonPart`; otherwise set the part to `kControlNoPart`. This example is somewhat trivial, in that the mouse down obviously occurred within the view bounds (the view would not receive the event otherwise), but if you have multiple parts within your view, you may need to perform several such tests to determine just what the user clicked.

12. Pass the part code back in the event by calling `SetEventParameter` .

    Later when the user performs additional mouse actions (mouse up, dragging, and so on), subsequent Carbon events sent to the view will have the part code parameter set to what you returned in the `kEventControlHitTest` handler.

13. Implement a mouse tracking handler. The system sends a `kEventControlTrack` event to your application when the user begins moving the mouse while holding the mouse button down. Your tracking handler must update the view depending on where the mouse goes.

14. Obtain the bounds of the window. Currently, the suggested mouse tracking function `TrackMouseLocation` returns the position of the mouse as a QuickDraw point. This means that you need to perform some calculations to convert the global QuickDraw point returned by `TrackMouseLocation` into the local coordinates of your custom view. Obtaining the bounds of the window containing the view enables you to translate the bounds later.

15. Obtain the current mouse location by calling `GetEventParameter`. If the mouse down occurred within the view, the first thing to do is to redraw the spot at that location.

16. If the part code indicates that the mouse action occurred within the view bounds, then change the spot position to be the current position of the mouse.

17. Constrain the spot to the drawn borders of the view. Use a series of conditionals to ensure that it can never be closer than 4 pixels from the actual view bounds.

18. Mark the view as needing to be redrawn by calling `HIViewSetNeedsDisplay`. On the next drawing pass, the draw handler draws the spot in the new location (that is, under the mouse).

19. Track the mouse by calling `TrackMouseLocation`. This function completes only when the user performs a mouse action (mouse up, drag, and so on), returning the action taken by the user and the current mouse position. Passing `-1` for the graphics port specifies that the mouse location should be returned in global coordinates. By keeping the `TrackMouseLocation` call within a loop, the mouse is continually tracked until the user releases the mouse button.

20. From the global coordinates of the mouse position, subtract off the top left position of the window. Doing so effectively translates the mouse position to be relative to the window's structure region (or root view).

21. Call `HIViewConvertPoint` to translate the point from the local coordinates of the root view to the local coordinates of your custom view.

Unlike the original custom user item, the custom view is not tied to a particular dialog; you need to define it only once and then you can use it in any window in your application. To add your view to a nib window, simply drag the HIView element into the window and specify its class ID and input parameters (if any) from the Info window. The view is instantiated automatically when you create the window from the nib file.

Document Revision History

This table describes the changes to *Upgrading to the Mac OS X HIToolbox*.

Date	Notes
2004-06-28	First public release.