HIToolbar Programming Guide

Carbon > Human Interface Toolbox

2005-07-07

Contents

Figures, Tables, and Listings

HIToolbar Concepts

Note: This document was previously titled *Using HIToolbar*.

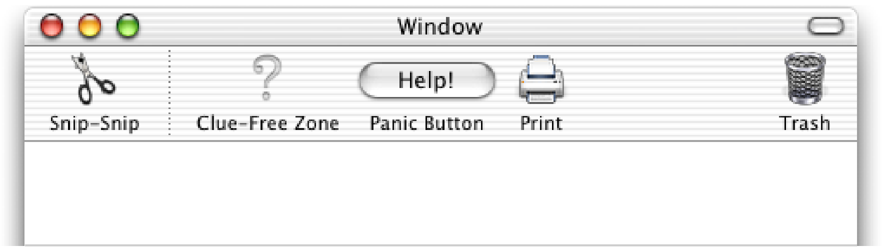
This chapter explains the concepts behind the Carbon HIToolbar. It describes the basic look and feel of a toolbar from a user's perspective, as well as "under-the-hood" details of how toolbars fit into the HIObject model.

Note: This document assumes you are familiar with the basics of Carbon event handling and manipulating HIObjects. If you are not familiar with these topics, see the documents *Carbon Event Manager Programming Guide* and *HIView Programming Guide*.

What Is a Toolbar?

A toolbar is a special container displayed directly underneath a window's title bar as shown in Figure 1-1. A toolbar can contain multiple toolbar items, which act as buttons or other controls. The user can show or hide the toolbar by clicking on the clear oblong "toolbar button" in the upper right corner of the window. A typical toolbar item displays an icon and is activated by a user click. While some items may simply mimic a simple push button, others may contain controls such as pop-up menus and search fields.

Figure 1-1 A toolbar



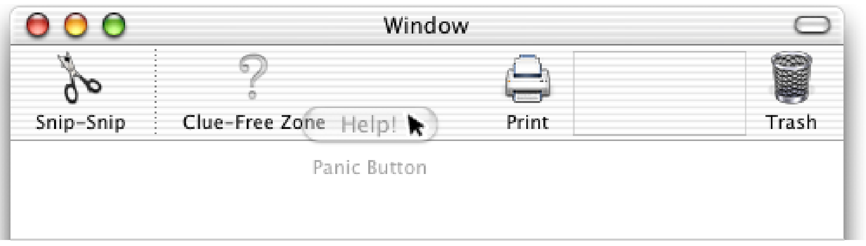
A toolbar can display its items as a combination of graphical image and text, text only, or graphics only, with two different sizes. The user can select view modes using a contextual menu displayed by control-clicking in the toolbar or cycle through all available modes by command-clicking the toolbar button. The user can also choose view modes while in the configuration sheet (as shown in Figure 1-3 (page 9).

The same toolbar can appear in multiple windows, providing a convenient way to access application features from a document. An application can also have multiple toolbars. For example, the Mail application has different toolbars for mail documents and viewer windows.

One advantage of a toolbar over a standard floating palette or other application-defined window is that the contents are user-configurable. The user has three ways to customize a toolbar's contents:

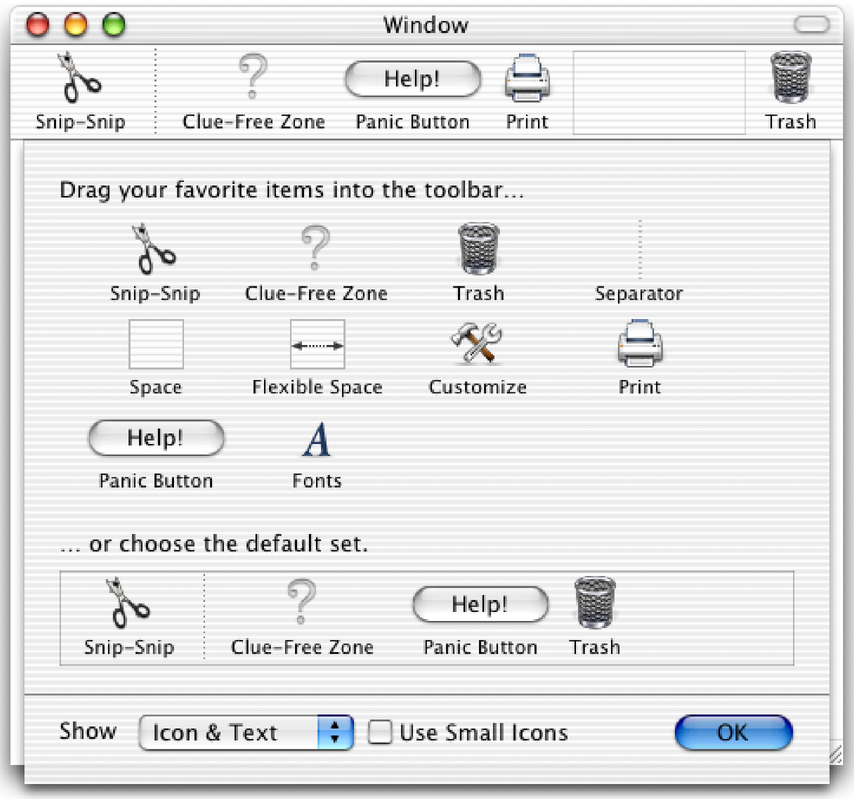
- The user can rearrange items in the toolbar (in Mac OS X v. 10.2.4 and later) by command-dragging them around, as shown in Figure 1-2. The user can also remove items entirely by command-dragging and releasing them outside the toolbar.

Figure 1-2 Dragging toolbar items



- If the toolbar supports drag-and-drop, the user may be able to create toolbar items by dragging specific data to the toolbar. For example, in Finder windows, a user can create folder items by dragging a folder into the toolbar. Another application could create a special URL item when the user drags text into the toolbar from a browser.
- If the toolbar is configurable, the user can bring up a sheet containing all the toolbar items supported by the application (excluding drag-and-drop items). The user can view the configuration sheet by choosing Customize Toolbar in the contextual menu brought up by control clicking in the toolbar or by command-option-clicking in the toolbar button.

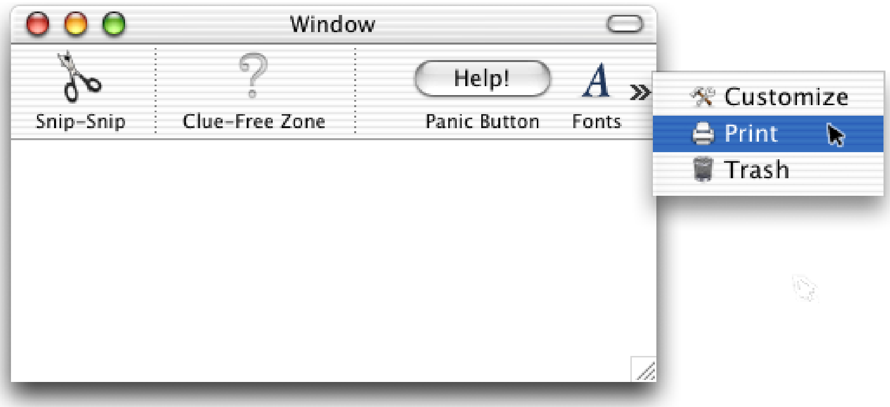
Figure 1-3 The toolbar configuration sheet



Changes made to the toolbar in one window automatically propagate to other windows sharing the same toolbar. The toolbar configuration is automatically stored in the user's Preferences folder when the application quits.

The toolbar can support any number of toolbar items. If there are more items than can be displayed in the window, the user can select the excess items using an overflow pop-up menu., as shown in Figure 1-4.

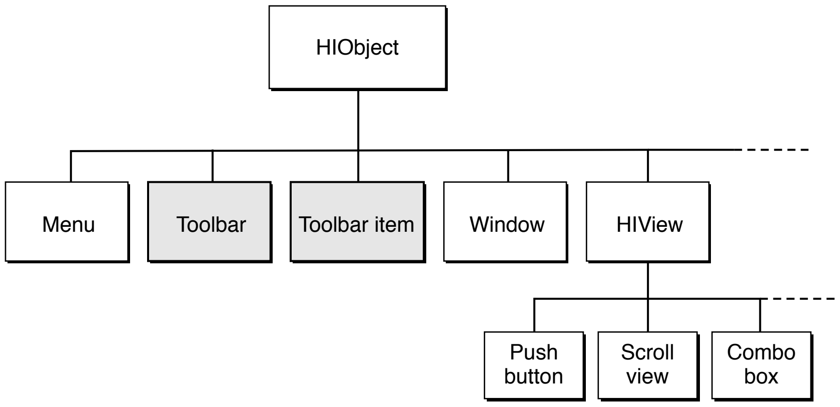
Figure 1-4 The toolbar overflow menu



The Toolbar as HIObject

In Carbon, the HIToolbar class is a subclass of HIObject (also introduced with Mac OS X v.10.2), which is the base class for all user interface objects in Carbon. Figure 1-5 (page 10) shows how toolbars and toolbar items fit into the HIObject class hierarchy.

Figure 1-5 The HIObject class hierarchy



The HIToolbar object is not associated with any window, and as such is not part of the standard event containment hierarchy. However, for each window in which a toolbar is to appear, the toolbar object creates toolbar views and toolbar item views and embeds them in that window's root view. These views are automatically updated when the toolbar contents change.

Model-View-Controller for HIToolbar

The Carbon toolbar follows the model-view controller convention as follows:

- The model is the list of toolbar items in a toolbar as well as the data each toolbar item displays or alters. In the HIToolbar API, these elements correspond to the HIToolbar and HIToolbarItem objects.
- The view is the visual representation of the toolbar and its items. A toolbar often has multiple views (one for each window that displays it), each of which shows a custom display of the HIToolbar model (depending on the size of the window and toolbar items, visibility of individual items, and so on). In the HIToolbar API, the view corresponds to the various HIViews created by the toolbar and its toolbar items.
- The controller is the code that determines how the view displays the model. For HIToolbar, all of the following correspond to the controller:
  - ❏ application-defined code that creates and manages views in custom toolbar items
  - ❏ code that modifies application data in response to clicks in toolbar items
  - ❏ code contained in any of the HIToolbar standard event handlers.

The Delegate

Associated with every HIToolbar object is a delegate, which is an event target containing a set of Carbon event handlers that enable the creation of toolbar items. While the toolbar is, by default, its own delegate, you can specify a different event target if you wish.

The delegate handles three basic requests, sent as Carbon events:

- A query for a list of default items. These are the initial items that appear in the toolbar.
- A query for a list of allowable items. This is the set of items that appear in the configuration sheet.
- A request to create a particular toolbar item.

Toolbar Items

Toolbar items are HIToolbarItem objects subclassed from HIObject. These objects are associated with HIToolbar objects, but are not embedded in the manner of HIViews.

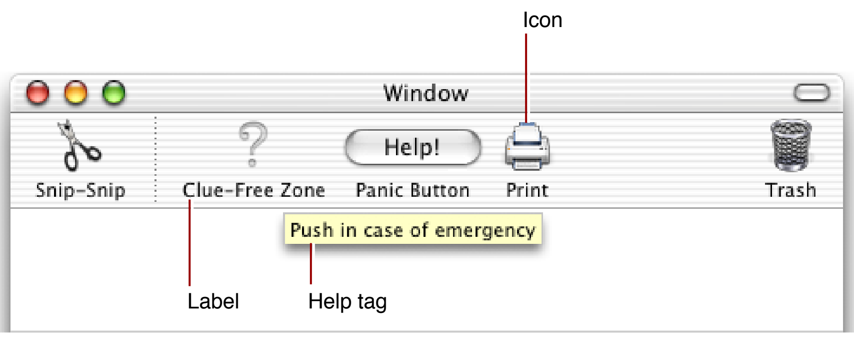
You identify toolbar items using a unique identifier. The system defines a number of standard toolbar items (such as Print, Customize, Separator, and so on) that you can create simply by specifying the proper identifier.

Most toolbar items act as buttons, performing a simple action when pressed. If you want a more sophisticated toolbar item, you can subclass the toolbar item HIObject class and implement your own behavior.

Toolbar items typically have the following data associated with them:

- An icon reference, which determines which icon to display for the item.
- A label, which appears below the icon, (and represents the item when in text-only mode).
- A help tag, which appears when the user hovers the cursor on the item for a few seconds.
- A command ID, which is sent in a `kEventCommandProcess` event when the user clicks on the toolbar item.

Figure 1-6 Toolbar item parts



Toolbar items also have attributes that govern their behavior. See "Toolbar Item Attributes" (page 18) for more information.

The HIToolbar object creates toolbar items in a lazy fashion. That is, no request is sent until the toolbar item needs to become visible.

Built-In Functionality

The toolbar is designed to be as easy to use as possible. To this end, much of the basic implementation is done for you. For example, doing any of the following requires no additional work on your part:

- Displaying the configuration sheet
- Saving the current toolbar setting to the user's preferences
- Showing and hiding the toolbar
- Drag tracking and drag rearranging of toolbar items
- Displaying contextual menus
- Moving items into the overflow menu when the toolbar runs out of space

In most cases all you need to worry about is creating the toolbar and any nonstandard toolbar items. The rest is done for you.

Note: Much of the "free" functionality comes from the standard window handler, so you should specify the standard handler for any window that contains a toolbar.

System Requirements

HIToolbar is available to Carbon applications in Mac OS X v.10.2 and later . Note that windows that contain toolbars do not have to support HIView compositing.

Toolbar Tasks

This chapter describes how to create and manipulate toolbars.

How the Toolbar Works

Before creating a toolbar, you should understand what goes on during the life of the HIToolbar object.

1. When the HIToolbar object is first created, it takes no additional actions until it actually becomes time to show a toolbar. For example, you can associate multiple windows with the toolbar, but it will not create any views until a toolbar must be displayed in a visible window.
2. When it is time to show a toolbar, the HIToolbar object creates a toolbar view and embeds it in the root view of the window that contains it.
3. The HIToolbar object then checks to see if any previously saved configuration information exists for this toolbar. If not, the HIToolbar object queries its delegate (through Carbon events) for the default configuration information.
4. For each item in the default configuration list, it sends a request to the delegate to create that toolbar item. The delegate should return a toolbar item reference for each item it creates. If the delegate returns `EventNotHandledErr`, the event passes to HIToolbar's default handlers, which will create any standard toolbar items.
5. For each toolbar item reference it receives, the HIToolbar object then creates a toolbar item view and embeds it in the toolbar view. If the toolbar item creates a custom HIView, this view is embedded in the toolbar item view.
6. HIToolbar repeats steps 2, 4, and 5 for each toolbar that appears in a visible window.
7. If the user changes the toolbar configuration in one window, the HIToolbar object is notified (using Carbon events) and it can then update the views in all the other windows associated with it.
8. If the user requests the Configuration sheet, the HIToolbar object queries its delegate for a list of allowable toolbar items. Using that list and the list of default items, it then requests toolbar items from the delegate to populate the Configuration sheet before displaying it.

Creating Toolbars

To create the toolbar, you simply call the `HIToolbarCreate` function. This creates an HIToolbar object, which controls the creation and display of the toolbar in associated windows. Listing 1-1 shows how you might create a toolbar and attach it to a window.

Listing 2-1 Creating a toolbar

```
OSStatus        err = noErr;
HIToolbarRef    toolbar;                                              // 1

err = HIToolbarCreate( CFSTR( "com.mycompany.carbontoolbar" ),        // 2
                    kHIToolbarAutoSavesConfig | kHIToolbarIsConfigurable,
                    &toolbar );


InstallEventHandler( HIObjectGetEventTarget( (HIToolbarRef)toolbar ),  // 3
                    ToolbarDelegate, GetEventTypeCount( kToolbarEvents ),
                    kToolbarEvents, toolbar, NULL );

SetWindowToolbar( window, toolbar );                                  // 4
ShowHideWindowToolbar( window, true, false );                         // 5

CFRelease( toolbar );                                                 // 6

ChangeWindowAttributes( window, kWindowToolbarButtonAttribute, 0 );   // 7

SetAutomaticControlDragTrackingEnabledForWindow( window, true );      // 8
```

Here is how the code works:

1. The toolbar is defined by a toolbar reference, which is also an HIObject reference.

2. To create a toolbar, call the `HIToolbarCreate` function. You must pass a unique identifer (as a Core Foundation string) for each toolbar your application creates. Doing so allows you to define multiple toolbars. The identifier should be in the form: com.*myCompany*.*myApp*.*myUniqueName*.

   You can specify attributes for your toolbar when you create it. `kHIToolbarAutoSavesConfig` indicates that the toolbar will save its current configuration to the application's user preferences file. The `kHIToolbarIsConfigurable` attribute indicates that the user can configure the toolbar (that is, the user can bring up the configuration sheet). If you do not want to specify any attributes, pass `kHIToolbarNoAttributes`.

3. You install the toolbar event handler on your delegate just as you would for any other event target. The delegate does all the event handling related to creating and managing toolbar items. When you first create the toolbar, the toolbar is its own delegate.

   To set a different delegate for the toolbar, call the `HIToolbarSetDelegate` function. Note that events are sent to the delegate with the `kEventTargetDontPropagate` option, so they cannot propagate up the event hierarchy.

4. To assign a toolbar to a window, call the `SetWindowToolbar` function. Note that you can assign the toolbar to multiple windows. Any window that contains the toolbar should use the standard window event handler.

5. The `ShowHideWindowToolbar` function shows or hides the toolbar (just as if the user pressed the toolbar button). The toolbar is initially hidden, so call this function to make it visible in the window.

6. Now that the toolbar is retained by a window, you can release your reference to it by calling `CFRelease`.

7. To add the oblong button to show and hide the toolbar, you must set the window attribute `kWindowToolbarButtonAttribute`. You can set this attribute within a nib file or when you programmatically create the window.

8. The toolbar requires automatic drag tracking, so you must set this by calling `SetAutomaticControlDragTrackingEnabledForWindow` on windows containing toolbars.

Event Handling Using the Delegate

After creating your toolbar, you need to assign an event handler to create and manage its toolbar items. These events are automatically sent to either the toolbar event target (the default) or an event target you specify by calling the `HIToolbarSetDelegate` function. In most cases, the toolbar itself is the appropriate delegate.

Listing 1-2 shows an example of a toolbar event handler.

Listing 2-2 A toolbar event handler

```
static const EventTypeSpec kToolbarEvents[] =                               // 1
{
    { kEventClassToolbar, kEventToolbarGetDefaultIdentifiers },
    { kEventClassToolbar, kEventToolbarGetAllowedIdentifiers },
    { kEventClassToolbar, kEventToolbarCreateItemWithIdentifier },
    { kEventClassToolbar, kEventToolbarCreateItemFromDrag }
};

static OSStatus ToolbarDelegate( EventHandlerCallRef inCallRef,
                                EventRef inEvent, void* inUserData )
{
    OSStatus            result = eventNotHandledErr;
    CFMutableArrayRef   array;
    CFStringRef         identifier;

    switch ( GetEventKind( inEvent ) )
    {
        case kEventToolbarGetDefaultIdentifiers:                            // 2
            GetEventParameter( inEvent, kEventParamMutableArray,
                            typeCFMutableArrayRef, NULL,
                            sizeof( CFMutableArrayRef ), NULL, &array );
            GetToolbarDefaultItems( array ); // application-defined
            result = noErr;
            break;

        case kEventToolbarGetAllowedIdentifiers:                            // 3
            GetEventParameter( inEvent, kEventParamMutableArray,
                            typeCFMutableArrayRef, NULL,
                            sizeof( CFMutableArrayRef ), NULL, &array );
            GetToolbarAllowedItems( array ); // application-defined
            result = noErr;
            break;

        case kEventToolbarCreateItemWithIdentifier:                         // 4
            {
                HIToolbarItemRef        item;
                CFTypeRef               data = NULL;

                GetEventParameter( inEvent,
                        kEventParamToolbarItemIdentifier, typeCFStringRef,
                        NULL, sizeof( CFStringRef ), NULL, &identifier );

                GetEventParameter( inEvent,
                        kEventParamToolbarItemConfigData, typeCFTypeRef,
                        NULL, sizeof( CFTypeRef ), NULL, &data );

                item = CreateToolbarItemForIdentifier( identifier, data );

                if ( item )                                                 // 5
                {
                    SetEventParameter( inEvent, kEventParamToolbarItem,
                            typeHIToolbarItemRef,
                            sizeof(HIToolbarItemRef ), &item );
                    result = noErr;
                }
            }
            break;

        case kEventToolbarCreateItemFromDrag:                               // 6
            {
                HIToolbarItemRef        item;
                DragRef                 drag;

                GetEventParameter( inEvent, kEventParamDragRef, typeDragRef,
                        NULL, sizeof( DragRef ), NULL, &drag );

                item = CreateToolbarItemFromDrag( drag );

                if ( item )
                {
                    SetEventParameter( inEvent, kEventParamToolbarItem,
                            typeHIToolbarItemRef,
                            sizeof( HIToolbarItemRef ), &item );
                    result = noErr;
                }
            }
            break;
    }

    return result;
}
```

Here is how the code works:

1. This example handler covers the four possible toolbar events:
    - A query for the default toolbar items. Default items are those contained in the toolbar when it first appears.
    - A query for allowable toolbar items. Allowable items are those that the user can place into the toolbar when in configuration mode. Note that if your toolbar is not configurable, you do not need to handle this event.

- A query to create a specific toolbar item. Each toolbar item has an unique identifier, which is passed in this event when the system wants to create it.
- A query to create a toolbar item based on a drag-and-drop action. For example, the Finder allows you to drag folders into the toolbar. If you want to add similar functionality to create folders, URLs, and so on from drags into your toolbar, you must handle this event.

2. When you receive the `kEventToolbarGetDefaultIdentifiers` event, you receive a pointer to a Core Foundation mutable array (obtained using the `kEventParamMutableArray` parameter). You must populate this array with the default toolbar item identifiers. The function to do so might look like this:

   ```
   static void GetToolbarDefaultItems( CFMutableArrayRef array )
   {
       CFArrayAppendValue( array, CFSTR( "com.apple.carbontoolbar.anchored" ) );
       CFArrayAppendValue( array, kHIToolbarSeparatorIdentifier );
       CFArrayAppendValue( array, CFSTR( "com.apple.carbontoolbar.permanent" ) );
       CFArrayAppendValue( array, kHIToolbarFlexibleSpaceIdentifier );
       CFArrayAppendValue( array, CFSTR( "MyCustomIdentifier" ) );
       CFArrayAppendValue( array, CFSTR( "com.apple.carbontoolbar.trash" ) );
   }
   ```

   The items appear in the toolbar in the order they are added to the array. In this default arrangement, the “anchored” item will appear on the far left, followed by the separator, and so on.

3. Similarly, when you receive the `kEventToolbarGetAllowableIdentifiers` event, you must populate a Core Foundation mutable array with the identifiers for the allowable toolbar items. These items appear in the configuration sheet in the order they are placed into the array. Note that you do not have to specify toolbar items that are created only from drag-and-drop actions.

4. When you receive a `kEventToolbarCreateItemWithIdentifier` event, you must create a toolbar item based on the identifier that was passed with the event. For example, after sending the `kEventToolbarGetDefaultIdentifiers` event, when the time comes to display the toolbar, the system sends a `kEventToolbarCreateItemWithIdentifer` event for each default item.

   The handler calls `GetEventParameter` to obtain the identifier (`kEventParamToolbarItemIdentifier`) and any associated configuration data that may be associated with it (`kEventParamToolbarItemConfigData`). For example, the configuration data could hold initial values for your item, a URL, text, and so on.

   You use the identifier to determine what kind of toolbar item to create. See “Creating Items from an Identifier” (page 19) for information about how you could implement the `CreateToolbarItemForIdentifier` function.

5. If your item creation function successfully created a toolbar item, call `SetEventParameter` to pass its reference back in the `kEventParamToolbarItem` parameter. Note that you are sent the `kEventToolbarCreateItemWithIdentifier` event even if the HIToolbar object wants to create a system-defined toolbar item. Therefore, whenever you don’t create a toolbar item, you should return `eventNotHandledErr` so the next handler in the calling chain has a chance to take the event.

6. When you receive a `kEventToolbarCreateItemFromDrag` event, you should create a toolbar item based on the user’s drag-and-drop action.

First, use GetEventParameter to obtain the drag reference (kEventParamDragRef) from the event. This example then passes the drag reference to its own toolbar item creation function. See "Creating Items from a Drag" (page 27) for information about how the CreateToolbarItemFromDrag function is implemented.

As with the kEventToolbarItemCreateItemWithIdentifier function, you pass the reference of the created toolbar item back into the event using SetEventParameter.

Creating Toolbar Items

In most cases, you create a toolbar item based on a unique identifier. The identifier is just that; a unique string that identifies a particular toolbar item. Identifiers you create are used only within your application, but they must be unique so that the system does not confuse them with identifiers from other applications.

System-Defined Identifiers

The HIToolbar API defines several standard identifiers that you can use in your application. Aside from returning the proper identifiers for the default and allowable item requests, you do not need to do anything else to implement these items.

Table 2-1 System-defined toolbar item identifiers

Identifier	Description
kHIToolbarSeparatorIdentifier	A thin vertical line used to group toolbar items.
kHIToolbarSpaceIdentifier	A fixed-width space.
kHIToolbarFlexible-SpaceIdentifier	A variable-width space.
kHIToolbarCustomizeIdentifier	Clicking on this item brings up the toolbar configuration sheet.
kHIToolbarPrintItemIdentifier	Clicking on this item sends a Print command event (just as if the user had selected the Print menu item) to the toolbar item view. You should install a command event handler at the window level to take this event.
kHIToolbarFontsItemIdentifier	Clicking on this item brings up the standard Fonts floating window. See the Apple Type Services (ATS) for Fonts documentation for details about how to set and obtain information from the Fonts window.

Toolbar Item Attributes

You can set attributes in your toolbar items that influence their behavior. Table 1-2 lists the available attributes.

Table 2-2 Toolbar item attributes

Attribute	Description
`kHIToolbarItemNoAttributes`	No attributes.
`kHIToolbarItemAllowDuplicates`	More than one item of this type can appear in the toolbar.
`kHIToolbarItemCantBeRemoved`	The user cannot drag this item out of the toolbar.
`kHIToolbarItemAnchoredLeft`	The item is fixed to the left side of the toolbar and cannot be moved. You can have multiple items with this attribute in a toolbar.
`kHIToolbarItemIsSeparator`	The item is used is a separator, and should appear as such in the overflow menu. Note that being a separator also implies that the item can draw the full height of the toolbar (unlike normal items, which typically have a label area below the graphic).
`kHIToolbarItemSend-CmdToUserFocus`	Any command events sent in response to clicks in this item are sent to the current user focus rather than the item's view.

Creating Items from an Identifier

This section describes how to create your toolbar items from identifiers passed to your application in the `kEventToolbarCreateItemWithIdentifier` event.

The toolbar sends this event to your application for every identifer, even those not defined by your application. Therefore, you should make sure that your delegate event handler returns `eventNotHandledErr` if you don't create a toolbar item.

Listing 1-3 shows a function used to create toolbar items.

Listing 2-3 Creating toolbar items from identifiers

```
static HIToolbarItemRef CreateToolbarItemForIdentifier(                  // 1
                            CFStringRef identifier, CFTypeRef configData )
{
    HIToolbarItemRef        item = NULL;

    if ( CFStringCompare( CFSTR( "com.apple.carbontoolbar.permanent" ),  // 2
                identifier, kCFCompareBackwards ) == kCFCompareEqualTo )
    {

        if ( HIToolbarItemCreate( identifier, kHIToolbarItemCantBeRemoved,
                            &item ) == noErr )                           // 3
        {
            IconRef     icon;
            MenuRef     menu;

            GetIconRef( kOnSystemDisk, kSystemIconsCreator, kFinderIcon, // 4
                            &icon );
            HIToolbarItemSetLabel( item, CFSTR( "Can't Remove Me" ) );   // 5
            HIToolbarItemSetIconRef( item, icon );                       // 6
            HIToolbarItemSetCommandID( item, 'shrt' );                      // 7
            ReleaseIconRef( icon );                                         // 8

            menu = NewMenu( 0, "\p" );                                      // 9
            AppendMenuItemTextWithCFString( menu, CFSTR( "Item 1" ), 0, 0,
                                 NULL );
            AppendMenuItemTextWithCFString( menu, CFSTR( "Item 2" ), 0, 0,
                                 NULL );
            AppendMenuItemTextWithCFString( menu, CFSTR( "Item 3" ), 0, 0,
                                 NULL );
            HIToolbarItemSetMenu( item, menu );                             // 10
            ReleaseMenu( menu );
        }
    }

    if ( CFStringCompare( CFSTR( "MyCustomIdentifier" ),                   // 11
                identifier, kCFCompareBackwards ) == kCFCompareEqualTo )
    {
        item = CreateMyButtonToolbarItem( CFSTR( "MyCustomIdentifier" ));
    }

    return item;
}
```

Here is how the code works:

1. This function, `CreateToolbarItemForIdentifier`, takes the two parameters passed to you in the `kEventToolbarCreateItemWithIdentifier` event.

2. Use the Core Foundation string compare function to determine which identifier is which. You specify the `kCFCompareBackwards` option to minimize the compare time.

3. If the identifier indicates the permanent toolbar item, create an unremovable toolbar item by calling the `HIToolbarItemCreate` function with the `kHIToolbarItemCantBeRemoved` attribute set. While a permanent toolbar item can be moved within the toolbar, the user cannot remove it.

4. You usually want to assign an icon to the toolbar item. In this case, you call the Icon Services function `GetIconRef` to obtain a system-defined icon (in this case, the Finder icon).

5. Call the `HIToolbarItemSetLabel` function to set the text that appears in the toolbar for this item.

6. Call the `HIToolbarItemSetIconRef` function to assign the icon reference you obtained for the toolbar item.

7. Call the `HIToolbarItemSetCommandID` function to assign a command ID to the toolbar item. Clicks on the item or its label will then generate a `kEventCommandProcess` event containing that command ID. This event is sent to the item's view unless you specified the `kHIToolbarItemSendCmdToUserFocus` attribute for your toolbar item.

   Note that if you do not assign a command ID to a toolbar item, the item is inherently not clickable, which may be preferable for certain custom items. For example, if the toolbar item displays a pop-up menu or similar control that has multiple states, you probably don't want the item's label to be clickable (except perhaps in text-only mode). Similarly, if a toolbar item contains an search field HIView, you want to take action based on what's typed into the view, not on clicks in the view or the item label.

Also, if you do not set a command ID for your toolbar item, that item will not be enabled when it appears in the overflow menu.

8. After you assign the icon reference to the toolbar item, you can release your reference to it.

9. If desired, you can add a pop-up menu to your toolbar item. This menu is used only for multi-state toolbar items when the toolbar is in text-only display mode. For example, the Finder's View toolbar item is a three-state button in icon view, corresponding to the three possible viewing modes (icon, list and column). In text-only mode, there is no button for the user to click, so the possible selections are made available using the pop-up menu.

   This example uses the Menu Manager function `NewMenu` to create a new empty menu, then calls the `AppendMenuItemTextWithCFString` to add menu items. You can then assign event handlers or command IDs to the menu to initiate actions when selected.

10. To assign the newly-created menu to the toolbar item, call the `HIToolbarItemSetMenu` function.

11. If the identifier indicates a custom ID, then create a custom toolbar item. For example, such a toolbar item could contain an HIView or other control, or it may be be created on the fly, from a drag-and-drop action. See "Creating Items from a Drag" (page 27) and "Embedding Views in a Toolbar Item" (page 21) for more information.

Creating Custom Toolbar Items

In some cases, you may want to create a special toolbar item that has nonstandard behavior. For example, you may want a toolbar item that is a slider, or a text input field, or you may want to create items from a drag-and-drop operation.

You create your custom toolbar item class by subclassing the toolbar item HIObject class and calling `HIObjectCreate` on that subclass. You need to register your subclass and provide handlers for construct, destruct, and initialize events, as well as some specific toolbar item–related events.

The sections that follow describe how to implement commonly-used custom toolbox items.

See *Inside Mac OS X: Introducing HIView* for more information about subclassing HIObjects and creating instances of those classes.

Embedding Views in a Toolbar Item

You can associate an HIView with a toolbar item. For example, you can place buttons, sliders, editable text fields, and other controls in the toolbar item. To do so, you must create a custom HIObject, specifically a subclass of the toolbar item class.

Note that, although the toolbar item is an HIObject, your view is not embedded within that object. For each window the toolbar is associated with, the toolbar item object creates a toolbar item view and then embeds your view within that.

Also, you do not have to supply your own item labels and help tags when creating your custom toolbar item; you can assign these as before using the `HIToolbarItemSetLabel` and `HIToolbarItemSetHelpText` functions.

To register your toolbar item subclass, you call the `HIObjectRegisterSubclass` function. The function in Listing 1-4 shows how you could do this.

Listing 2-4 Registering a toolbar item subclass

```
#define kMyButtonToolbarItemClassID CFSTR("com.moof.buttontoolbaritem")

void RegisterMyButtonToolbarItemClass()
{
    static bool sRegistered;

    if ( !sRegistered )
    {
        HIObjectRegisterSubclass( kMyButtonToolbarItemClassID,
                kHIToolbarItemClassID, 0, MyButtonToolbarItemHandler,
                GetEventTypeCount( buttonEvents ), buttonEvents, 0, NULL );

        sRegistered = true;
    }
}
```

This example registers a new toolbar item subclass (`kMyButtonToolbarItemClassID`) and its associated event handler `MyButtonToolbarItemHandler`. This toolbar item simply contains a standard push button.

For efficiency, this example also sets a static Boolean to avoid registering the custom item multiple times (although there is no penalty for doing so).

The `MyButtonToolbarItemHandler` must handle the HIObject construct, initialize, and destruct events. In addition, because you want to embed a view into the toolbar item view, you must also handle the `kEventToolbarItemCreateCustomView` event.

Listing 1-5 shows how you might implement the event handler for your custom button toolbar item.

Listing 2-5 Event handler for a toolbar item with an embedded view

```
const EventTypeSpec buttonEvents[] =
{
    { kEventClassHIObject, kEventHIObjectConstruct },
    { kEventClassHIObject, kEventHIObjectInitialize },
    { kEventClassHIObject, kEventHIObjectDestruct },

    { kEventClassToolbarItem, kEventToolbarItemCreateCustomView }
};
const EventTypeSpec pushButtonEvents[] =
{
    { kEventClassControl, kEventControlGetSizeConstraints}
};
struct MyButtonToolbarItem                                        // 1
{
    HIToolbarItemRef        toolbarItem;
};
typedef struct MyButtonToolbarItem MyButtonToolbarItem;

static OSStatus MyButtonToolbarItemHandler( EventHandlerCallRef inCallRef,
                             EventRef inEvent, void* inUserData )
{
    OSStatus            result = eventNotHandledErr;
    MyButtonToolbarItem*    object = (MyButtonToolbarItem*)inUserData;        // 2

    switch ( GetEventClass( inEvent ) )
    {
        case kEventClassHIObject:
            switch ( GetEventKind( inEvent ) )
            {
                case kEventHIObjectConstruct:                                  // 3
                    {
                        HIObjectRef         toolbarItem;
                        MyButtonToolbarItem*    item;

                        GetEventParameter( inEvent,
                             kEventParamHIObjectInstance, typeHIObjectRef,
                             NULL, sizeof( HIObjectRef ), NULL,
                             &toolbarItem );

                        result = ConstructMyButtonToolbarItem(toolbarItem,
                             &item );

                        if ( result == noErr )
                            SetEventParameter( inEvent,
                                kEventParamHIObjectInstance, typeVoidPtr,
                                sizeof( void * ), &item );
                    }
                    break;

                case kEventHIObjectInitialize:                                 // 4
                    if (CallNextEventHandler(inCallRef, inEvent) == noErr )
                        {
                            HIToolbarItemSetLabel( object->toolbarItem,
                                                    CFSTR( "Press Me" ) );
                            HIToolbarItemSetHelpText( object->toolbarItem,
                                                    CFSTR("Moof!"), NULL );
                            result = noErr;
                        }
                    break;

                case kEventHIObjectDestruct:                                   // 5
                    free (object);
                    result = noErr;
                    break;
            }
            break;

        case kEventClassToolbarItem:
            switch ( GetEventKind( inEvent ) )
            {
                case kEventToolbarItemCreateCustomView:                        // 6
                    {
                        EventTargetRef myButtonEventTarget;
                        HIViewRef myButton;
                        Rect myButtonRect = {0,0,20,80};

                        CreatePushButtonControl(NULL, &myButtonRect,           // 7
                             CFSTR("Push!"), &myButton);

                        SetEventParameter (inEvent, kEventParamControlRef,
                              typeControlRef, sizeof(myButton), &myButton);

                        myButtonEventTarget =
                                        GetControlEventTarget(myButton);

                        InstallEventHandler (myButtonEventTarget,              // 8
                                MyButtonEventHandler,
                                GetEventTypeCount(pushButtonEvents),
                                pushButtonEvents, NULL, NULL);
                        result = noErr;
                    }
                    break;
            }
            break;
    }

    return result;
}
```

Here is what the code does:

1. Your subclass must have a structure associated with it that can hold any necessary instance data. This example requires only only one data item: the toolbar item reference. Note that even though this toolbar item provides an HIView, you do not store the HIView reference in this structure.

2. If your application defines any user data for this toolbar item, you can store it in the instance data.

3. When you receive the `kEventHIObjectConstruct` event, you receive a pointer to store the reference to your toolbar item in the `kEventParamHIObjectInstance` parameter. You must allocate memory for the item reference and any associated instance data. Here is a possible implementation for the `ConstructMyToolbarButtonItem` function:

   ```
   static OSStatus ConstructMyButtonToolbarItem( HIToolbarItemRef inItem,
                                   MyButtonToolbarItem** outItem )
   {
       MyButtonToolbarItem*        item;
       OSStatus                err = noErr;

       item = (MyButtonToolbarItem*)malloc( sizeof( MyButtonToolbarItem ) );
       require_action( item != NULL, CantAllocItem, err = memFullErr );

       item->toolbarItem = inItem;

       *outItem = item;

       CantAllocItem:
       return err;
   }
   ```

   After allocating the memory, use `SetEventParameter` to return a pointer to the space created for the toolbar item reference.

4. When you receive the `kEventHIObjectInitialize` event, you must perform any initialization that your toolbar item requires, such as setting initial values for some instance data. This example calls the `HIToolbarSetItemLabel` function to set the text portion of the toolbar item, and `HIToolbarItemSetHelpText` function to set the help tag text.

5. When you receive the `kEventHIObjectDestruct` event, you must release the memory that was allocated for your toolbar item. This example simply calls the Memory Manager function `free` to release the instance data. Note that you must not dispose your HIView reference here.

6. When you receive the `kEventToolbarItemCreateCustomView`, you must create an HIView to embed in the toolbar item view. Any sort of view is valid; you can embed a picture view, a text field, a button, or even your own custom HIView.

7. This example simply creates a standard push button control and then uses `SetEventParameter` to store its HIView reference in the `kEventParamControlRef` parameter of the event. Note that when creating standard views, you must pass `NULL` for the owning window parameter, as the view is not associated with any window.

8. All views that are associated with toolbar items must respond to the `kEventControlGetSizeConstraints` event, which asks for the maximum and minimum allowable sizes for the view. The system uses this information to determine how to best size the toolbar item view given its embedded content. In most cases you should simply return the bounds of the view, as in the following code:

```
static OSStatus MyButtonEventHandler( EventHandlerCallRef inCallRef,
                         EventRef inEvent, void* inUserData )
{

switch ( GetEventKind( inEvent ) )
            {
                case kEventControlGetSizeConstraints:
                    {
                        HISize minBounds = {80,20};
                        HISize maxBounds = {80,20};

                        SetEventParameter (inEvent,
                                 kEventParamMinimumSize, typeHISize,
                                 sizeof(HISize), &minBounds);
                        SetEventParameter (inEvent,
                                 kEventParamMaximumSize, typeHISize,
                                 sizeof(HISize), &maxBounds);

                        return noErr;
                    }
                    break;

                default:
                    break;
 }
return eventNotHandledErr;
}
```

Note that the `HISize` type is position-independent, indicating only the width and height of the object.

Finally, you need to call `HIObjectCreate` to create an instance of your custom toolbar item. Listing 1-6 shows a function you could use to do this.

Listing 2-6 Creating a custom toolbar item instance.

```
HIToolbarItemRef CreateMyButtonToolbarItem( CFStringRef inIdentifier)
{
    OSStatus            err;
    EventRef            event;
    UInt32              options = kHIToolbarItemAllowDuplicates;
    HIToolbarItemRef    result = NULL;

    RegisterMyButtonToolbarItemClass();                               // 1

    err = CreateEvent( NULL, kEventClassHIObject,                     // 2
                kEventHIObjectInitialize, GetCurrentEventTime(), 0,
                &event );
    require_noerr( err, CantCreateEvent );

    SetEventParameter( event, kEventParamToolbarItemIdentifier,       // 3
            typeCFStringRef, sizeof( CFStringRef ), &inIdentifier );
    SetEventParameter( event, kEventParamAttributes, typeUInt32,      // 4
            sizeof( UInt32 ), &options );

    err = HIObjectCreate( kMyButtonToolbarItemClassID, event,         // 5
                        (HIObjectRef*)&result );
    check_noerr( err );

    ReleaseEvent( event );                                            // 6

CantCreateEvent:
    return result;
}
```

Here is how the code works:

1. First, register your custom toolbar item by calling the `RegisterMyButtonToolbarItemClass` function in Listing 1-4 (page 22).

2. Create an initialization event to send to your item once it is created. Note that this event is sent directly to the toolbar item (that is, it is not placed in the event queue).

3. Set the toolbar item's identifier in the initialization event using the `kEventParamToolbarIdentifier` parameter.

4. Set any desired toolbar attributes in the initialization event using the `kEventParamAttributes` parameter.

   Note that the identifier and the toolbar attributes are the same parameters you would pass into the standard `HIToolbarItemCreate` function. If you do not set these parameters, the `HIObjectCreate` call fails.

5. Call `HIObjectCreate` to create an instance of your custom toolbar item class (`kMyButtonToolbarClassID`). On return, you receive a `ToolbarItemRef` (which is also an `HIObjectRef`) that points to the newly-created toolbar item.

6. After sending the initialization event to your toolbar item, you no longer need to retain it, so call `ReleaseEvent`.

Creating Items from a Drag

If the user drags something onto the toolbar, and your toolbar accepts drags, your application will have to create a toolbar item from the drag.

Listing 1-7 shows how you could process the drag to create a toolbar item. This example creates URLs from one or more text drag items.

Listing 2-7 Creating a toolbar item from a drag

```
static HIToolbarItemRef CreateToolbarItemFromDrag( DragRef drag )
{
    UInt16              i, itemCount;
    HIToolbarItemRef    result = NULL;

    CountDragItems( drag, &itemCount );                                 // 1

    for ( i = 1; i <= itemCount; i++ )
    {
        DragItemRef         itemRef;
        FlavorFlags         flags;

        GetDragItemReferenceNumber( drag, i, &itemRef );                // 2

        if ( GetFlavorFlags( drag, itemRef, 'TEXT', &flags ) == noErr )  // 3
        {
            Size        dataSize;
            char        string[256];
            CFURLRef    url;

            dataSize = sizeof( string );
            GetFlavorData( drag, itemRef, 'TEXT', &string, &dataSize, 0 );  // 4

            url = CFURLCreateWithBytes( NULL, string, dataSize,          // 5
                                kCFStringEncodingMacRoman, NULL );

            result = CreateMyURLToolbarItem( CFSTR( "MyURLIdentifier" ), url );// 6

            CFRelease( url );                                            // 7

            break;
        }
    }

    return result;
}
```

Here is how the code works:

1. This example uses a number of Drag Manager calls to process the drag item. First, you call the `CountDragItems` function on the drag reference to determine how many items are in the drag. (For example, the user might have selected and dragged several items into the toolbar.)

2. Now iterate over the number of drag items. For each item index number, obtain the item reference by calling `GetDragItemReferenceNumber`.

3. Use the Drag Manager function `GetFlavorFlags` to determine if the drag item content is of type `TEXT`. If so, the item contents can be turned into a URL toolbar item. If not, the drag item is rejected and the contents ignored.

4. If the drag flavor is compatible, call the Drag Manager function `GetFlavorData` to obtain the actual text associated with the drag item.

5. The Core Foundation function `CFURLCreateWithBytes` converts the string into a CFURL with the specifed encoding (in this case, MacRoman).

6. Now pass the CFURL to a toolbar item creation function. This function, which is analogous to the `CreateMyButtonToolbarItem` function in Listing 1-6 (page 25), should register the toolbar item subclass, create a `kEventHIObjectInitialize` event, and call `HIObjectCreate`.

   You should set several parameters in the initialize event before calling `HIObjectCreate`:

   - `kEventParamToolbarIdentifier`, the identifier for this toolbar item.
   - `kEventParamAttributes`, any toolbar attributes you want to set.
   - `kEventParamToolbarItemConfigData`, a pointer to any specific configuration data. In this case, the configuration data would be a pointer to the URL obtained from the drag.

7. After passing the CFURL to the toolbar item creation function, you can release your reference to it.

The event handler for your custom URL toolbar item class must handle the usual creation, destruction, and initialization events. The construction event allocates memory for the instance data, and the destruction event releases it. Note that the data structure for the instance data has two fields in this case: one to hold the toolbar item reference, and the other to hold the URL.

```
struct CustomToolbarItem
{
    HIToolbarItemRef        toolbarItem;
    CFURLRef                url;
};
typedef struct CustomToolbarItem CustomToolbarItem;
```

The initialization event handler must obtain the configuration data passed with the event and store that in the instance data as a URL. Listing 1-8 shows an example of how you could do this.

Listing 2-8 Initialization function for the kEventHIObjectInifialize event

```
/* Assume that the event handler called CallNextEventHandler before
/* calling this function */

static OSStatus InitializeCustomToolbarItem( CustomToolbarItem* inItem,        // 1
                     EventRef inEvent )
{
    CFTypeRef       data;
    IconRef         iconRef;

    if ( GetEventParameter( inEvent, kEventParamToolbarItemConfigData,         // 2
                     typeCFTypeRef, NULL,
            sizeof( CFTypeRef ), NULL, &data ) == noErr )
    {
        if ( CFGetTypeID( data ) == CFStringGetTypeID() )                      // 3
```

```
            inItem->url = CFURLCreateWithString( NULL, (CFStringRef)data,
                                NULL );
        else
            inItem->url = (CFURLRef)CFRetain( data );
    }
    else
    {
        inItem->url = CFURLCreateWithString( NULL,                        // 4
                            CFSTR( "http://www.apple.com" ), NULL );
    }

    HIToolbarItemSetLabel( inItem->toolbarItem, CFSTR( "URL Item" ) );      // 5

    if ( GetIconRef( kOnSystemDisk, kSystemIconsCreator, kGenericURLIcon,   // 6
                &iconRef ) == noErr )
    {
        HIToolbarItemSetIconRef( inItem->toolbarItem, iconRef );
        ReleaseIconRef( iconRef );                                          // 7
    }

    HIToolbarItemSetHelpText( inItem->toolbarItem,                          // 8
                    CFURLGetString( inItem->url ), NULL );

    return noErr;
}
```

Here is how the code works:

1. This function takes two parameters: the toolbar item to initialize, and the initialization event that was sent to it.

2. Use `GetEventParameter` to obtain the configuration data stored in the initalization event. This is the URL obtained from the drag (the one stored in the `kEventHIObjectInitialize` event passed to `HIObjectCreate`.

3. If the configuration data passed is of type `CFString`, then you must call `CFURLCreateWithString` to convert it to a `CFURL` type before storing it in the instance data structure.

   If the data is already of type `CFURL`, we can simply retain its reference and store it in the instance data.

4. If for some reason no configuration data was passed in the initialization event, set the instance data URL to some default value.

5. Call `HIToolbarItemSetLabel` to set the displayed text for the toolbar item.

6. If your custom toolbar item does not create an HIView to display, in most cases you should assign it an icon. This example simply uses the Icon Services function `GetIconRef` to obtain a generic URL icon, and assigns it to the toolbar item by calling `HIToolbarItemSetIconRef`.

7. After assigning the icon reference to the toolbar item, you can release your reference to it.

8. Call `HIToolbarItemSetHelpText` to assign the help tag text for your toolbar item.

In addition to the `kEventClassHIObject` events, your custom URL toolbar item should also handle two additional events:

- `kEventToolbarItemGetPersistentData`: You receive this event when the toolbar wants to store configuration information in the user's preferences. You should return a pointer (which must be of type `CFTypeRef`) to any data you want to store in the `kEventParamConfigData` parameter.

  Note that this data is saved in XML format; any data you store in this event must be CFPropertyList-compatible. Currently this means the data must be one of the following types: `CFString`, `CFData`, `CFNumber`, `CFBoolean`, `CFDate`, `CFArray`, and `CFDictionary`.

  Note that for CFType references, you should pass a copy of the data, as the system will dispose of the reference it receives to the configuration data.

- `kEventToolbarItemPerformAction`: You receive this event when the user clicks on your toolbar item. Your handler can then take any appropriate action. For example, if the user clicks on your custom URL toolbar item, you can call the Launch Services function `LSOpenCFURLRef` to open the URL using the user's default browser.

  Note that if your toolbar item handles the `kEventToolbarItemPerformAction` event, you should set the command ID for the toolbar item to `kHIToolbarCommandPressAction` (value: `'tbpr'`). Doing so ensures that HIToolbar can send the perform action event when your toolbar item is selected from the overflow menu. This command ID is available in Mac OS X v.10.2.3 and later.

Document Revision History

This table describes the changes to *HIToolbar Programming Guide*.

Date	Notes
2005-07-07	Changed title from "Using HIToolbar." Made minor bug fix.
2003-04-30	In "What Is a Toolbar?" (page 7), indicated that the user can cycle between display modes by command-clicking the toolbar button.
	Added info in "What Is a Toolbar?" (page 7) describing how the user can bring up the configuration sheet.
	Added overflow menu screen shot, Figure 1-4 (page 9).
	Added new section "Model-View-Controller for HIToolbar" (page 10) showing how HIToolbar fits into the model-view-controller design convention.
	Added command ID as one of the data types associated with toolbar items in "Toolbar Items" (page 11).
	Removed `kHIToolbarItemValidAttrs` and `kHIToolbarItemMutableAttrs` from "Toolbar Item Attributes" (page 18), as these constants may change over time.
	In "Creating Items from an Identifier" (page 19), indicated that toolbar items that do not have a command ID associated with them will not be enabled when they appear in the overflow menu.
	Added info to "Creating Items from an Identifier" (page 19) describing cases where you might not want to assign a command ID to a toolbar item.
	Replaced `'void'` with `typeVoidPtr` in Listing 1-5 (page 22) in the `SetEventParameter` call for the construct event.
	Added specific information in "Creating Items from a Drag" (page 27) about what kind of data you can store in response to the `kEventToolbarItemGetPersistentData` event.
	Indicated in "Creating Items from a Drag" (page 27) that if you handle the `kEventToolbarItemPerformAction` event, you must also set the command ID of the toolbar item to `kHIToolbarCommandPressAction`.
	Removed Panther-specific information.
2003-03-01	Preliminary review draft.