Accessibility Programming Guidelines for Carbon

Carbon > Accessibility

2007-02-08

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

Simultaneously published in the United States and Canada.

Contents

Figures

Introduction to Accessibility Programming Guidelines for Carbon

All Carbon applications can and should be accessible to users with disabilities. The process of making an application accessible is called access enabling. How big a job this is depends on the extent to which your application uses custom user interface objects.

If your Carbon application relies on HIObjects for all its user interface elements (including subclasses of HIView), most of the accessibility infrastructure is provided for you. If, on the other hand, your Carbon application uses some custom subclasses of HIObject or HIView or relies on a custom application framework, you need to supply more of the accessibility infrastructure yourself.

This document outlines how to access-enable applications throughout this range. It provides steps you can follow to access-enable an application that uses only HIObjects and HIViews in its user interface. It then provides guidelines to help you access-enable an application that implements custom views or depends on a custom application framework.

Who Should Read This Document

All Carbon application developers should read this document to learn how to make their applications accessible to users with disabilities. If you're new to accessibility you should read *Accessibility Overview* to get an overview of the Mac OS X accessibility architecture.

If you're an assistive application developer, you don't need to read this document. Instead, you should read *Accessibility Overview* to become familiar with the Mac OS X accessibility architecture and then you should read *Accessibility Reference for Assistive Applications*.

Organization of This Document

This document has the following chapters:

- "Accessibility and the Carbon Framework" (page 9) describes how Carbon implements accessibility and provides support for access-enabling Carbon applications.
- "Making a Standard Carbon Application Accessible" (page 17) describes the steps you follow to access-enable a Carbon application that uses only standard HIObjects.
- "Making a Semistandard Carbon Application Accessible" (page 23) describes additional steps you follow to access-enable a Carbon application that implements some custom subviews.
- "Making a Custom Carbon Application Accessible" (page 29) provides guidelines to help you access-enable a Carbon application that implements a custom view subsystem or implements its interface procedurally.
- "Revision History" (page 33) describes changes to this document.

See Also

The Accessibility Reference Library contains several documents that cover accessibility:

- *Getting Started with Accessibility* provides a brief introduction to accessibility and describes learning paths you might choose to follow.
- *Accessibility Overview* describes the Mac OS X accessibility architecture.
- *Accessibility Programming Guidelines for Cocoa* describes how to access-enable a Cocoa application.
- *Accessibility Reference for Assistive Applications*
- *Carbon Accessibility Reference* describes the functions, data types, and constants used in accessible Carbon applications.
- NSAccessibility describes the NSAccessibility protocol and its methods and constants.

In addition to these documents, Apple maintains a website devoted to accessibility in Mac OS X, with links to more information about compatible assistive technologies:

- http://www.apple.com/accessibility

Accessibility and the Carbon Framework

The Mac OS X accessibility architecture defines the model that applications follow to make themselves accessible to assistive applications and technologies. The model is not tied to any one application framework, so each framework is free to implement accessibility support in the most natural and efficient way.

This chapter describes how the Carbon framework implements accessibility for Carbon applications in Mac OS X. If your application uses only standard HIObject objects and subclasses to implement its user interface, you should skim this chapter for background information before you read "Making a Standard Carbon Application Accessible" (page 17).

If you implement some custom subclasses of HIObject or if your application is based on a custom application framework (such as PowerPlant), you should read this chapter to enhance your understanding of the Carbon accessibility implementation. Then, you should read "Making a Semistandard Carbon Application Accessible" (page 23) or "Making a Custom Carbon Application Accessible" (page 29) to learn how to access-enable your application.

If you're unfamiliar with the Mac OS X accessibility model or you're unsure why your application should be accessible, read *Accessibility Overview* for an introduction to accessibility in Mac OS X.

Accessibility Objects in Carbon

As described in *Accessibility Overview*, the Mac OS X accessibility architecture models the user interface of an accessible application as a hierarchy of accessibility objects. Each of these objects represents an accessible user interface object in the application, such as a button or a window. The accessibility object provides information about itself (such as its capabilities, position in the accessibility hierarchy, and a localizable description) to assistive applications. An assistive application uses this information to tell the user about the application's features, help the user navigate, and perform the application's functions.

The following sections describe the basics of the Carbon implementation of an accessibility object. If your application implements custom subclasses of HIObject or HIView or uses a custom application framework, you should read these sections to become familiar with how Carbon defines the accessibility object.

The Carbon Implementation of the Accessibility Object

Carbon bases its implementation of the accessibility object on the HIObject subsystem. Introduced in Mac OS X version 10.2, HIObject is the base class for all user interface objects in Carbon. If you're unfamiliar with HIObject and its subclass HIView (the base class for all view-based controls), you should read *Upgrading to the Mac OS X HIToolbox* and *HIView Programming Guide*.

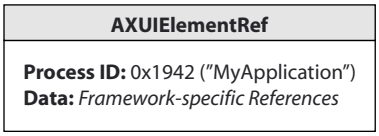
In Carbon, an accessibility object is an `AXUIElementRef`. An `AXUIElementRef` is a Core Foundation data structure that contains two members:

- The HIObject reference of an accessible user interface object, such as a window or a button

- A 64-bit identifier that can be used to represent an accessible subcomponent of a complex user interface object, such as a window, a button, or some other object of your design

Figure 2-1 (page 10) shows the structure of the accessibility object as it is implemented in Carbon.

Figure 1-1 The Carbon accessibility object



Carbon implements an accessibility object as a `CFTypeRef` object because this allows it to be flattened easily and sent across processes. This is important because the Carbon events that carry accessibility messages both send and receive accessibility objects. It also allows you to put accessibility objects inside another `CFTypeRef` object, such as a CFArray. An event handler does this, for example, when it responds to the request for all accessible children of a given accessibility object.

The HIObject inside the accessibility object is significant both because it represents the user interface object and because it is a target for Carbon events related to that object. Recall from *Accessibility Overview* that in Mac OS X, accessibility is message-driven. An assistive application communicates with an application by sending messages to the application's user interface. In a Carbon application, these messages take the form of Carbon events. Because an HIObject is automatically a target for Carbon events, the accessibility object containing the HIObject is also a target. In addition, the Carbon accessibility implementation installs the accessibility-specific event handlers on all standard HIObjects and HIViews. For more information on the role of Carbon events in the Carbon accessibility implementation, see "Accessibility Carbon Events" (page 12).

The accessibility object identifier provides an efficient way for Carbon to represent the subcomponents of a complex object without needing multiple HIObjects. Consider a data browser object that displays check boxes and relevance indicators in addition to the list and column views of its data. For an assistive application to access every check box, relevance indicator, and item of data, each of these components must be represented by an accessibility object. Rather than create a separate HIObject for each accessibility object, however, Carbon uses the HIObject reference of the parent object coupled with a unique identifier for each component. By convention, an identifier with the value 0 means the accessibility object represents the object as a whole (the parent object), rather than one of the object's components.

Because the accessibility object uses the HIObject reference of a user interface object, it is very lightweight. In addition, because the core of the accessibility object is the HIObject to which it refers, its life span can be very short. In general, an accessibility object is created, returned in an event parameter, and released. Although you could create accessibility objects and reserve them for later use, there is little to no performance penalty for creating them as needed and then releasing them.

If your application uses only standard HIObjects and subclasses in its user interface, it is unlikely that you will need to pay much attention to the value of an accessibility object's identifier. If you override the standard event handlers on these objects, however, you may have to use the identifier to determine which child of the object a given accessibility object represents.

If you implement custom subclasses of HIObject or HIView or use a custom application framework, you can use the accessibility object identifier to represent the containment hierarchy of your complex objects in your own way. If you do this, be sure to follow the convention of giving the top-level, parent object an identifier with the value 0. Then, you can give the child objects identifiers that make sense in your implementation.

Creation of Accessibility Objects

Carbon does not create an accessibility object for every standard HIObject in an application when it launches. Because accessibility is message-driven, it is not necessary to create large numbers of accessibility objects before they are required. Instead, Carbon creates an accessibility object for a user interface object only when necessary, in response to an accessibility Carbon event. This is a technique you can use if you handle events that require you to return accessibility objects for your custom subclasses of HIObject or custom user interface objects.

In general, if your application uses only HIObject and HIView objects in its user interface, you create very few accessibility objects. About the only accessibility object you have to create in a standard Carbon application is one to represent a static text title you display near a control or view. (To learn how to do this, see "Link Objects and Related Static Text Titles" (page 19).) If you implement custom subviews or other user interface objects, however, you must create accessibility objects for them when an assistive application needs to access them.

When you create an accessibility object, you use the `AXUIElementCreateWithHIObjectAndIdentifier` function (defined in `CarbonEvents.h`). This function requires an HIObject reference and an identifier and it returns a reference to the accessibility object. Note that because Carbon implements the accessibility object as a `CFTypeRef` object, the normal Core Foundation semantics apply. In particular, you must use `CFRelease` to dispose of the accessibility object when it is no longer needed.

If your application uses a custom application framework, your user interface objects probably are not implemented as HIObjects. When you create an accessibility object for such an object, you first instantiate an HIObject wrapper for it and then create the accessibility object. For more information on how to do this, see "Making a Custom Carbon Application Accessible" (page 29).

Accessibility Object Attributes

When Carbon creates an accessibility object to represent a given HIObject or HIView, it also provides some default values for the required attributes, based on the type of the object. It does this in response to a Carbon event requesting the value of a given attribute. For a button, for example, Carbon can supply values for attributes such as the role, the size, and the view that contains the button. Carbon gets most of this information from the object implementation, but context-specific attribute values must be supplied by the application.

The most important attribute value an application supplies is the accessibility object's description. Carbon cannot supply the value of a description attribute, because it is specific to the way an application uses an object. If your application contains a print button, for example, and you do not supply a description, an assistive application can tell a user only that the object is a button, not that it is a print button.

If your application uses only standard HIObject and HIView objects, there are only a few attribute values you must supply. For the most part, these attributes describe the layout and specific features of your application. To learn how to supply these values, see "Making a Standard Carbon Application Accessible" (page 17).

When you create an accessibility object for a custom object, you must supply additional attribute values. Because the set of required and optional attributes is largely dictated by the role of the accessibility object, the first thing you have to do is determine what role the accessibility object should have. It is best to leverage the existing accessibility protocol and choose one of the roles Apple defines to describe the new accessibility object. For more information about the set of roles and the attributes associated with them, see the appendix "Roles and Associated Attributes" in *Accessibility Overview*.

Parameterized Attributes

In Mac OS X version 10.3, the Carbon accessibility implementation introduced parameterized accessibility object attributes. A parameterized attribute is a special attribute that allows an assistive application to access individual parts of an accessibility object.

Used mainly for objects that contain user-entered text, a parameterized attribute can refer to something as specific as a particular line within an editable text area. Unless your application handles complex, editable text areas, you probably do not need to use parameterized attributes.

Ignored UIElements

As described in *Accessibility Overview*, an application can simplify its accessibility hierarchy by designating selected accessibility objects as ignored. In general, these accessibility objects represent implementation-specific parts of the user interface that a user does not access. Because an assistive application is interested in only accessible objects, ignored objects are not included in the accessibility hierarchy an assistive application sees. It's worthwhile to identify accessibility objects that should be ignored because doing so streamlines the user interface.

Carbon provides the `HIObjectSetAccessibilityIgnored` function (defined in `HIObject.h`) you can use to adjust the ignored status of an accessibility object. This function requires an accessibility object and a Boolean value that indicates whether or not the accessibility object should be ignored. This function sets the ignored status of the passed-in accessibility object alone; it does not touch the accessibility object's descendents. If you want an accessibility object's descendents to be ignored as well, you must call this function on each of them. By default, a newly created accessibility object is not ignored.

It's important to note that an accessibility object's ignored status is not something your application needs to take into account during its normal processing. In particular, you should continue to return accessibility objects when appropriate, regardless of their ignored status. For example, if you handle an event that requests an accessibility object's child, you return that child whether or not it is ignored. The Carbon accessibility implementation automatically removes ignored accessibility objects from the accessibility hierarchy it presents to assistive applications.

Accessibility Carbon Events

In `CarbonEvents.h`, Carbon defines ten accessibility Carbon events of the event class `kEventClassAccessibility`. An accessible application responds to these events, giving assistive applications information about specific accessibility objects and performing requested actions.

The following sections give you an overview of the accessibility Carbon events and describe the part they play in accessibility. For reference documentation on these events, see *Carbon Accessibility Reference*. If you're unfamiliar with the Carbon event mechanism, you should read *Carbon Event Manager Programming Guide* for an introduction.

Note: For the sake of simplicity, these sections discuss the accesssibility Carbon events as being "sent by" assistive applications. In fact, a nonCarbon assistive application does not actually send Carbon events. Instead, it sends messages the Carbon framework transmits as Carbon events.

If you use only standard HIObject and HIView objects in your application's user interface and you're using Mac OS X version 10.4 or later, you probably do not have to create any custom accessibility event handlers. This is because, as described in "The Carbon Implementation of the Accessibility Object" (page 9), Carbon automatically installs the necessary event handlers on standard HIObjects and HIViews. However, if you implement any custom user interface objects (or subviews) or if you need to handle an event in a nonstandard way, you should read this section to learn about the events you might have to handle.

In addition, all developers should read "Providing Attribute Values Without Event Handlers" (page 15). This section describes how a new function introduced in Mac OS X version 10.4 allows you to set selected accessibility object attribute values without having to create custom event handlers.

Data Conversion in Carbon Accessibility Events

Carbon supports automatic runtime data conversion of accessibility event parameters. For example, in response to a request for the value of an accessibility object's position attribute, you can return an `HIPoint` type value. Carbon automatically converts this value into the expected `kAXValueCGPointType` value.

This allows you to handle values in the best way for your application without worrying about how the Carbon accessibility implementation uses them.

Hierarchy-Based Events

The parent-child relationships among accessibility objects define an application's accessibility hierarchy. As described in *Accessibility Overview*, an assistive application views your application through its accessibility hierarchy. It is therefore essential that all accessible objects in your application appear in their proper positions in the application's accessibility hierarchy.

The accessibility hierarchy is based on the containment hierarchies defined by the user interface objects. For a Carbon application, this means the containment hierarchies of HIObject and HIView objects. The default HIView subsystem can accurately represent an HIView's containment hierarchy as long as all subviews of the view are also HIViews. The handlers Carbon installs on these objects automatically supply the appropriate accessibility hierarchy information. If you do not implement any custom subviews (and you do not redefine any parent-child relationships), you do not have to provide your own handlers for the accessibility hierarchy events.

Two of the accessibility hierarchy-based events concern hit-testing and keyboard focus:

- `kEventAccessibleGetChildAtPoint`
- `kEventAccessibleGetFocusedChild`

The hit-testing event, `kEventAccessibleGetChildAtPoint`, contains an accessibility object and a global screen point. Standard HIObjects and HIViews automatically handle this event by returning the accessibility object's immediate child accessibility object that contains the point (if one exists).

An assistive application sends the keyboard-focus event, `kEventAccessibleGetFocusedChild`, to find the focused child of the given accessibility object. The default handler determines if an immediate child of the accessibility object is in the focus chain. If such a child exists, the handler returns it.

Both these events might seem to encourage recursive hit-testing and focus-testing on the part of the handler receiving the event, but this not the case. The Carbon accessibility implementation performs the necessary recursion, so it is important that the handlers for these events do not also perform recursive testing.

Carbon does not define a separate event an assistive application can use to get the accessible parent or children of an accessibility object. Instead, an assistive application sends the `kEventAccessibleGetNamedAttribute` event, specifying the parent or children attribute. Standard HIObject and HIView handlers return the requested value, which an assistive application can use to traverse the accessibility hierarchy. For more information about this event, see "Attribute Events" (page 14).

Action Events

Carbon defines three accessibility action events:

- `kEventAccessibleGetAllActionNames`
- `kEventAccessibleGetNamedActionDescription`
- `kEventAccessiblePerformNamedAction`

An assistive application must be able to drive an application's user interface just as a user does with the mouse or keyboard. As part of this process, an assistive application sends the `kEventAccessibleGetAllActionNames` event to discover which actions an accessibility object can perform. The handler for this event adds the names of the object's supported actions to the array contained in one of the event parameters.

Some assistive applications need to speak the name of an action to the user. When this is the case, the assistive application sends the `kEventAccessibleGetNamedActionDescription` event. Even though the name of this event contains the word "Description", don't confuse this with an accessibility object's description attribute. Whereas the description attribute is an application-specific description of the accessibility object's purpose, the action description is simply its human-intelligible, localizable name. The `AXActionConstants.h` file defines the Mac OS X accessibility actions, along with their names. If you need to supply an event handler for this event, you must supply the action description value. See "Handle Action Events" (page 27) for information on how to do this.

An assistive application sends the `kEventAccessiblePerformNamedAction` event to transmit the user's request to perform an action. The handler for this event makes sure the accessibility object supports this event and, if so, tells the application to perform it.

Attribute Events

An assistive application relies on an accessibility object's attribute values to tell it how to handle the object the accessibility object represents. An assistive application may also set the values of some attributes in response to a user's actions. Carbon defines five accessibility events that allow an assistive application to query, get, and set attribute names and values:

- `kEventAccessibleIsNamedAttributeSettable`

- kEventAccessibleGetAllAttributeNames
- kEventAccessibleGetAllParameterizedAttributeNames
- kEventAccessibleGetNamedAttribute
- kEventAccessibleSetNamedAttribute

An assistive application sends the kEventAccessibleIsNamedAttributeSettable event to find out if the given attribute value is settable. Settable attribute values are those that can change dynamically while the application runs. An example of a settable attribute is the value attribute of an editable text field.

If you provide an event handler to return an attribute's value, you must also provide a handler for the kEventAccessibleIsNamedAttributeSettable event. Your handler returns the settability of the specific attribute for which you provide a value. This way, you ensure that an assistive application gets the correct information about this attribute, instead of relying on the default handler to provide it.

An assistive application uses the kEventAccessibleGetAllAttributeNames event to request the names of all an accessibility object's attributes *except* parameterized attributes. To request the names of parameterized attributes, an assistive application sends the kEventAccessibleGetAllParameterizedAttributeNames event. The handlers for both events return an array of attribute names, represented as CFStrings.

When an assistive application needs to get the value of a specific attribute, it sends the kEventAccessibleGetNamedAttribute event. The handler for this event returns the value of the specified attribute, whether or not it is a parameterized attribute.

Providing Attribute Values Without Event Handlers

In versions of Mac OS X prior to 10.4, the only way to provide application-specific values of accessibility object attributes is to install a custom handler that provides those values. For example, to provide the value of an accessibility object's description attribute, you install a handler that returns to the kEventAccessibleGetNamedAttribute event a CFString containing the description.

Mac OS X version 10.4 introduced the HIObjectSetAuxiliaryAccessibilityAttribute function (defined in HIObject.h) that you can use in place of an event handler to provide values for static, unsettable attributes. An attribute value is settable if a user (through an assistive application) can directly change it, such as the selected text attribute of an editable text field object. A good way to think of this is, if an attribute helps describe the layout of your user interface, such as a description or a title, its value is probably static and unsettable.

When you use the HIObjectSetAuxiliaryAccessibilityAttribute function to set attribute values, it holds those values in a store that the HIObject event handler at the bottom of the handler stack can consult. When no higher-level handlers can handle the kEventAccessibleGetNamedAttribute event for a specific attribute, the event passes to the HIObject handler, which consults the store for the requested attribute value. This scheme gives you the flexibility to set some attribute values as your application launches and others by installing custom handlers.

It's important to note that you don't have to use this function to set the values of static attributes. Depending on other accessibility-related tasks you perform, it might be more efficient to install event handlers to provide these values, even if you're developing in Mac OS X version 10.4.

Accessibility Notifications

As described in *Accessibility Overview*, an accessibility object can broadcast notifications about changes in its state. Assistive applications register for these notifications so they can keep track of the creation, change in value, and destruction of accessibility objects.

The Carbon accessibility implementation defines many notifications in `AXNotificationConstants.h` (in the HIServices framework). It also defines the `AXNotificationHIObjectNotify` function you use to send notifications for an accessibility object. If your application uses only standard, noncustom HIObjects and HIViews, all necessary notifications are sent for you. If, on the other hand, you implement custom user interface classes and, consequently, create accessibility objects to represent them, you have to send your own notifications. This is particularly important for the value-change notifications. If you implement a complex user interface object (one that has its own children), only you know which child view's value has changed. For more information on sending notifications, see "Making a Semistandard Carbon Application Accessible" (page 23).

Key Modifiers and VoiceOver

In Mac OS X version 10.4, Apple introduced VoiceOver, a state-of-the-art, built-in screen reading interface to the Macintosh. Users can activate VoiceOver from System Preferences (in the Universal Access pane) or by pressing Command-F5. VoiceOver is a prime example of an assistive application that uses the accessibility information your application provides to present a verbal picture of your application to the user.

When using VoiceOver, the Option and Control keys are considered VoiceOver control keys. VoiceOver does not hide the use of these keys when it is telling an application to perform an action, such as open a document or push a button. This is not a problem unless an application uses the `GetCurrentEventKeyModifiers` function to discover which modifier keys are in an event. Contrary to its name, the `GetCurrentEventKeyModifiers` function returns the modifiers that were in the *last* user event that reached the application. It does not return the modifiers that are used at the time the event is being processed. If your application uses this function while VoiceOver is running, it will interpret the Option key as a modifier of the primary key, instead of as the VoiceOver trigger. This is wrong and will result in behavior the user does not desire.

If your application uses the `GetCurrentEventKeyModifiers` function, you should replace it with code that extracts the desired information from the event parameter. This will allow you to discover the modifiers used in key-down and mouse-down events without impairing the accessibility of your application.

Making a Standard Carbon Application Accessible

Most new Carbon applications rely on HIToolbox technologies to implement their user interfaces. This chapter defines a standard Carbon application as one that fits the following criteria:

- It uses only standard HIObjects and HIViews to represent all user interface objects.
- It uses only standard HIViews to represent all subviews of a complex HIView, such as the contents of a tabs view.
- The accessibility hierarchy of the application does not deviate from the containment hierarchy defined by the HIObject and HIView objects in the user interface. In other words, you do not redefine the parent-child relationships among user interface objects.

If this describes your application, read this chapter to learn how to access-enable your application. If your application fits most of the criteria, but implements some custom behavior, read this chapter first. Then, read "Making a Semistandard Carbon Application Accessible" (page 23) to learn how to handle the custom behavior.

How Much Work Will This Be?

Because your application uses only standard HIObjects and subclasses in its user interface, there is very little you have to do to make it accessible. Carbon provides much of the accessibility infrastructure, so your task is mainly one of enhancement and refinement. This is because the Carbon accessibility implementation can handle generic things such as window placement, containment hierarchies, and actions, but it can't predict or discover many of the application-specific details that make your application unique. Far from being merely cosmetic, therefore, the information you provide not only makes your application more accessible, it enhances the user's experience.

The rest of this chapter describes specific steps you take to make your application accessible. In addition, be sure your application does not use the `GetCurrentEventKeyModifiers` function. If it does, see "Key Modifiers and VoiceOver" (page 16) for information about why you should not use this function and a suggestion for how to replace it.

Note: Some of the following sections describe how to provide attribute values using the `HIObjectSetAuxiliaryAccessibilityAttribute` function introduced in Mac OS X version 10.4. You can use this function to supply the value of any static, nonsettable attribute. If you must supply the value of a settable or dynamically determined attribute or you are using an earlier version of Mac OS X, you can use the alternate technique to set attribute values described in "Install the Necessary Event Handlers" (page 21).

Provide Descriptive Information for All Elements

An assistive application needs to be able to describe all accessible user interface elements to the user. Often, an assistive application can present the title of the element to the user, but sometimes an element's title is either unavailable or not sufficiently descriptive. Therefore, you must examine your application and ensure that all accessibility objects supply descriptive information about themselves in either the title or description attributes.

First, determine if a given accessibility object already includes the title attribute. A standard HIView object that displays a text title as part of its visual interface, such as an OK button, already includes the title attribute with the value of the displayed text. Such an object does not need a description attribute because its title is sufficient to convey its purpose to the user. (If a user interface element is titled by a separate static-text string, see "Link Objects and Related Static Text Titles" (page 19) for details on how to provide this information to assistive applications.)

A button that displays an icon instead of a text title, however, does not have a title attribute. An example of such an object is a "back" button that displays a left-pointing arrow instead of the word "Back". An assistive application cannot describe such a button's purpose to a user unless the accessibility object representing the button includes the description attribute. If you have such an object in your application, you must supply an appropriate description in the description attribute.

An accessibility object's description attribute is static and not settable by an assistive application. For this reason, you can use the `HIObjectSetAuxiliaryAccessibilityAttribute` function to set it. If you're interested in how this function works, see "Providing Attribute Values Without Event Handlers" (page 15). If you're developing in a version of Mac OS X prior to 10.4, see "Install the Necessary Event Handlers" (page 21) to learn how to set the description value with an event handler.

If you must supply the value of an accessibility object's description attribute, be sure that the description:

- **Is accurate and succinct**. Remember that a user may hear this description spoken aloud by an assistive application every time the object has keyboard focus or is under the mouse pointer.
- **Does not include the role of the object**. An assistive application, such as a screen reader, is likely to speak a concatenation of the description string you provide and the role description value. If you include the role in your description, such as "left justify button", a screen reader may speak "left justify button button".
- **Uses only lower case and does not include any punctuation**. This ensures that your description is intelligible to the largest range of assistive applications. Setting a description value to "left justify" instead of "left-justify", for example, avoids potential confusion.

Link Objects and Related Static Text Titles

It is not uncommon for an application to display a piece of static text that serves as the title for a user interface object. You might choose to do this if you want to display a title for some number of user interface objects that do not display titles as part of their visual display. If this is something you do in your application, you must make the relationship between the static text and the interface object or objects clear to assistive applications. Otherwise, an assistive application cannot tell a user that the static text title and the user interface object or objects are related.

Mac OS X version 10.4 introduced two new attributes to handle this situation:

- `kAXTitleUIElementAttribute`
- `kAXServesAsTitleForUIElementsAttribute`

Because these attributes are static and nonsettable (they help define the layout of your application), you can use the `HIObjectSetAuxiliaryAccessibilityAttribute` function to provide their values. As their names imply, the values of these attributes are not strings, but accessibility objects. This means that you must create a new accessibility object to represent the static text before you use the `HIObjectSetAuxiliaryAccessibilityAttribute` function to link it to the user interface object (or set of objects) it describes.

The value of the `kAXTitleUIElementAttribute` attribute is an accessibility object that represents a static text title. The value of the `kAXServesAsTitleForUIElementsAttribute` attribute is an array of accessibility objects. This allows you to link several user interface objects to a single title.

To see how this works, consider a set of editable text fields intended to contain parts of a mailing address. Instead of titling each text field separately, you supply a single static text title, such as “Address:”, to title the set of text fields. To link these text fields with the static text title, you follow these steps:

1. Use the `AXUIElementCreateWithHIObjectAndIdentifier` function to create an accessibility object to represent the static text title (“Address:”, in this example).

2. Add the `kAXServesAsTitleForUIElementsAttribute` attribute to the accessibility object representing the static text title that you created in Step 1. The value of this attribute is an array containing the accessibility object (or objects) representing the user interface object (or objects) you want to be associated with this static text title.

   In this example, the value of the `kAXServesAsTitleForUIElementsAttribute` attribute is an array containing the accessibility objects that represent the editable text fields.

3. Add the `kAXTitleUIElementAttribute` attribute to the accessibility object representing the user interface object to which you want to relate the title. The value of this attribute is the accessibility object you created in Step 1 to represent the static text title.

   In this example, you add the `kAXTitleUIElementAttribute` attribute to each accessibility object that represents one of the editable text fields.

Provide Labels

A label is separate static text that gives additional information about parts of a control. For example, an application might display a slider with labeled tick marks that describe the slider's measurements.

To a sighted user, the proximity of the label and the slider implies their relationship. To an assistive application, however, the label and the slider are unrelated. To allow an assistive application to provide this information to the user, therefore, you must specify the relationship between them.

Unlike the title of a button, a label is not part of the user interface object it describes. Instead, a label is similar to a static text title, such as the title "Address:" in the example in "Link Objects and Related Static Text Titles" (page 19).

In the example above, the slider is accompanied by a set of tick marks, each of which refers to one value in the range of values the slider can have. Each tick mark is considered to be a separate label. This allows you to provide such information in discrete pieces, instead of in a single string. Such a slider does not have a label consisting of the single string "0 5 10 15 20", therefore, but a set of labels, each of which contains a single element, such as "0", "5", "10", "15", and "20".

To provide such information to an assistive application, you create an accessibility object for each label. Each of these accessibility objects includes a `kAXLabelValueAttribute` attribute that contains the value of that label. The accessibility object representing the control (the slider in this example) contains a `kAXLabelUIElementsAttribute` attribute. The value of this attribute is an array of accessibility objects, which allows you to link an arbitrary number of labels with the control.

Link Related Views

If your application displays different views of the same or related content, you must make this relationship clear to assistive applications. For example, if you display a list of files in one window and you allow the user to view the contents of a selected file in another part of the window or in a different window, an assistive application cannot know that the two views are related. Similarly, an assistive application cannot predict where you display the results of a search field in your application. You must tell an assistive application how such views are related to ensure that a user can move directly between them without having to step through every intervening view and control.

To make this relationship explicit, add the `kAXLinkedUIElementsAttribute` attribute to each related view's accessibility object. The value of this attribute is an array of accessibility objects so you can specify different configurations of related views, such as one-to-one and one-to-many.

You can provide these links either statically or dynamically, depending on the design of your application. If a link is dictated by the layout of your application, you can use the `HIObjectSetAuxiliaryAccessibilityAttribute` function to provide the attribute values. If, on the other hand, a link is determined dynamically when a user uses your application, you set the value of the `kAXLinkedUIElementsAttribute` attribute with an event handler for the `kEventAccessibleGetNamedAttribute` event.

Install the Necessary Event Handlers

The `HIObjectSetAuxiliaryAccessibilityAttribute` function introduced in Mac OS X version 10.4 gives you a convenient way to provide the values of static attributes. If you are developing in a version of Mac OS X prior to 10.4, however, or you are dealing with settable attribute values, you use the Carbon event mechanism to set them. For an overview of the Carbon accessibility events, see "Accessibility Carbon Events" (page 12).

Setting Attribute Values

To set the value of a settable attribute, create a handler for the `kEventAccessibleSetNamedAttribute` event. This event includes parameters that contain the name of the attribute and the new value. Your handler should ensure that:

- The accessibility object supports the given attribute.
- The data (which can be of an arbitrary Core Foundation type) makes sense for the attribute.

If the handler accepts the given parameter values, it should set the value of the attribute to the passed-in data. If the accessibility object does not support the given attribute, the handler should return the `eventNotHandledErr` error.

Providing Attribute Values

It's worth repeating that most Carbon applications that do not implement any custom user interface objects or custom subviews of HIView do not have to install custom event handlers. For such applications, the standard event handlers Carbon installs should be sufficient.

To provide an accessibility object's attribute value using an event handler, create a handler for the `kEventAccessibleGetNamedAttribute` event. This event includes a parameter that contains the name of the attribute (in a CFString object). Your handler should ensure that the accessibility object supports the attribute. If it does, the handler should return the value of the attribute in the *`kEventParamAccessibleAttributeValue`* parameter. If it does not support the attribute, your handler should return the `eventNotHandledErr` error.

The `kEventAccessibleGetNamedAttribute` event can also include a parameter that describes a part of a parameterized attribute. As described in "Parameterized Attributes" (page 12), parameterized attributes became available in Mac OS X version 10.3. If the given accessibility object supports parameterized attributes, your handler should return the value for the part of the element specified by the *`kEventParamAccessibleAttributeParameter`* parameter. Again, if the accessibility object does not support the attribute, your handler should return the `eventNotHandledErr` error.

If you're developing an application in a version of Mac OS X prior to 10.4, you must install a custom event handler to provide the value of the description attribute for those accessibility objects that need it. Your handler must also provide the values of the title and label attributes if you support them. This event handler responds to the `kEventAccessibleGetNamedAttribute` event by returning the value of the passed-in attribute in the *`kEventParamAccessibleAttributeValue`* parameter.

Making a Semistandard Carbon Application Accessible

Many applications use HIObject and HIView objects, but may implement custom classes to represent some of the contents of those objects. An example is an application that creates a custom HIView to represent a custom control not provided by the system, such as a volume indicator control.

This chapter defines a semistandard Carbon application as one that fits the following criteria:

- It uses only HIObject and HIView objects to represent its user interface objects, although it may use custom classes to implement substructure.
- It is Carbon event driven.
- The accessibility hierarchy may deviate from the containment hierarchy defined by the HIObjects and HIViews.

If this describes your application, there are a few things you must do to make sure assistive applications can access all parts of your application's user interface. This chapter describes the steps you take to access-enable the custom portions of your application. It does not describe the steps you take to access-enable the standard portions of your application. If you haven't already, be sure to read "Making a Standard Carbon Application Accessible" (page 17) to learn how to provide the context-specific attribute values all applications must supply.

How Much Work Will This Be?

Although access enabling a semistandard Carbon application is somewhat more work than access enabling a standard Carbon application, the task is quite manageable. In general terms, you need to provide the same information about your custom objects that Carbon provides about standard objects. Specifically, this means you:

- Create an accessibility object for each accessible custom subview of the HIView objects in the user interface, if necessary.

  For each accessibility object you create, you must provide the necessary attributes and make sure it is properly inserted into your application's accessibility hierarchy.
- Adjust the default accessibility hierarchy, if necessary, to accommodate child objects you might add or remove.
- Install any custom Carbon accessibility event handlers your application requires to implement custom behavior.

The following sections describe how to accomplish these steps. To learn how to provide application-specific values for descriptions, titles, and labels, see "Making a Standard Carbon Application Accessible" (page 17).

In addition, be sure your application does not use the `GetCurrentEventKeyModifiers` function. If it does, see "Key Modifiers and VoiceOver" (page 16) for information about why you should not use this function and a suggestion for how to replace it.

Create Accessibility Objects for Custom Subviews

A single HIView object can represent a simple user interface element that contains no substructure or a complex element that contains a great deal of substructure. A data browser, for example, is a single HIView that contains a complex substructure. As described in "The Carbon Implementation of the Accessibility Object" (page 9), Carbon can represent the accessible parts of a complex substructure with a single HIView combined with a unique identifier.

In your application, you might define both simple and complex custom HIView objects. For these objects to be accessible, you must create accessibility objects to represent them. The following sections describe how to do this.

Creating an Accessibility Object for a Simple HIView

If your custom subview is an HIView and it contains no substructure, create an accessibility object for it with an identifier value of `0`. Recall from "The Carbon Implementation of the Accessibility Object" (page 9) that an identifier value of `0` indicates that the accessibility object represents the user interface element as a whole. Because this accessibility object represents a user interface element that has no substructure, it automatically represents the element as a whole.

To create an accessibility object for such a user interface element, use the `AXUIElementCreateWithHIObjectAndIdentifier` function, passing in the HIViewRef reference and 0, as shown below:

```
accessibilityObject = AXUIElementCreateWithHIObjectAndIdentifier( HIObjectRef
myHIView, 0 )
```

Because your custom subview is an HIView, Carbon provides attribute values it can extract from the object, such as size and position. As with other accessibility objects in your application, however, you must provide the context-specific information that Carbon cannot determine. For information on providing descriptive and linking information, see "Making a Standard Carbon Application Accessible" (page 17). If there are additional attribute values you need to provide, do this with custom event handlers. For more information on how to do this, see "Install Custom Event Handlers" (page 26).

Creating an Accessibility Object for a Complex HIView

Although most custom HIViews an application might create are uncomplicated, it's not uncommon for an application to need a custom HIView with complex substructure. From an accessibility perspective, however, such an HIView is accessible but its contents are not. This is because Carbon does not automatically create accessibility objects for the children of a custom HIView.

If you do this in your application, the first thing you need to do is create an accessibility object to represent each accessible child of the HIView. As mentioned in "Creation of Accessibility Objects" (page 11), you may choose to create these accessibility objects only when needed (in response to an assistive application's

requests) or in advance (when you open the application or display a window). Either way, when it's time to provide an accessible subview, use the `AXUIElementCreateWithHIObjectAndIdentifier` function to create an accessibility object for it.

As described in "The Carbon Implementation of the Accessibility Object" (page 9), an accessibility object consists of an HIObject and an identifier. If you are creating an accessibility object for a child view of an HIView, for example, pass to the `AXUIElementCreateWithHIObjectAndIdentifier` function:

- The `HIViewRef` of the parent object
- An integer you define that uniquely identifies the subview

As with other accessibility objects in your application, you must provide the context-specific descriptive and linking information that Carbon cannot determine. For information on providing this information, see "Making a Standard Carbon Application Accessible" (page 17). To provide additional attribute values, use custom event handlers, as described in "Install Custom Event Handlers" (page 26).

Adjusting the Accessibility Hierarchy

If you add a new subview to an existing HIView you need to adjust the accessibility hierarchy that Carbon creates. Similarly, you must adjust the accessibility hierarchy if you want to prevent a subview of an HIView from being accessible. This section describes how to adjust the accessibility hierarchy to accommodate these changes. This section assumes that you have already created accessibility objects to represent any new subviews you are adding to the hierarchy. If you have not, see "Create Accessibility Objects for Custom Subviews" (page 24) for more information on how to do this.

To add the accessibility object representing a subview to your application's accessibility hierarchy, you follow these steps:

1. Make sure the accessibility object you create returns its new parent when asked for the value of the parent attribute.

   In most cases, the parent of the new accessibility object is its containing HIView. However, there may be times when you want an accessibility object to be the child of some more distant ancestor. Be sure to map out these relationships to ensure their correct representation in the accessibility hierarchy.

2. Install on the parent object an event handler for the `kEventAccessibleGetNamedAttribute` event. When an assistive application asks for the value of the children attribute, your handler gets the child array the parent object's event handler provides and adds the new child (or children) to it. Your handler then returns the modified array.

3. Override the parent object's hit-testing and keyboard-focus event handlers so you can return more specific information.

It is unlikely you will ever need to remove a child accessibility object from the accessibility hierarchy. If you do, however, the process is simple:

1. Install on the parent object an event handler for the `kEventAccessibleGetNamedAttribute` event. When an assistive application asks for the value of the children attribute, your handler gets the child array the parent object's event handler provides and removes the child from it. Your handler then returns the modified array.

2. Override the parent object's hit-testing and keyboard-focus event handlers so you can remove the child from the return values.

3. Inspect the accessibility hierarchy for other attributes that might refer to the child you want to remove. A convenience attribute, such as the kAXHorizontalScrollBarAttribute attribute, might refer to a child you want to remove. If you find any such attributes, adjust your handler for the `kEventAccessibleGetNamedAttribute` event to handle these, as well.

Install Custom Event Handlers

Although the Carbon accessibility implementation automatically handles events for standard HIViews, some of these events require information about the children of an HIView. If you implement these children, you must install custom handlers that provide information about them.

The types of event you must handle depend on the complexity of the subviews you implement.

- **Simple subview**. This is a subview that does not contain any children. For these subviews, you must make sure the parent HIView includes the subview in its children attribute array. Then, you can rely on the parent's event handlers to return accurate information about your custom subview, including hit-testing and keyboard-focus information.
- **Complex subview**. This is a subview that contains an arbitrarily complex hierarchy of children. For these subviews, you must make sure the parent HIView includes your top-level subview in its children attribute array. Additionally, you must install handlers on your subview and its children to handle other events, such as hit-testing and keyboard-focus events.

Handle Events for Simple Subviews

If you use a custom class to implement a simple subview of an HIView, your main task is to make sure the parent view is aware of its child view. This ensures that your subview accessibility object is part of the accessibility hierarchy. Then, you may choose to provide handlers for other events.

To ensure that the parent HIView is aware of your simple subview, you must insert the accessibility object you've created for the subview into the parent's children attribute array. To do this, you install on the parent HIView a handler for the `kEventAccessibleGetNamedAttribute` event that listens for a request for the value of the children attribute. When such a request arrives, your custom handler gets the parent HIView's children attribute array and inserts your accessibility object.

After your custom accessibility object is a member of its parent's children attribute array, you can rely on the parent's default hit-testing and keyboard-focus handlers to return your subview when appropriate.

Handle Events for Complex Subviews

If you implement a complex subview, you must handle the `kEventAccessibleGetNamedAttribute` event both for your parent HIView (as described in "Handle Events for Simple Subviews" (page 26)) and for your top-level subview. In addition, you must handle the hit-testing and keyboard-focus events that request information about your subview's children.

To report information about your subview's internal structure, install on your subview's accessibility object a handler for the `kEventAccessibleGetNamedAttribute` event. Your custom handler responds to the request for the value of the children attribute by creating and returning an array of the subview's accessible child accessibility objects. If your complex subview represents several levels of containment, you have to provide a similar handler for each level.

To handle the hit-testing and keyboard-focus events for your subview's accessible children, install handlers for the `kEventAccessibleGetChildAtPoint` and `kEventAccessibleGetFocusedChild` events. It is important to return only the immediate child of the accessibility object receiving these events, not a grandchild or more distant descendant. This is because the Carbon accessibility implementation automatically handles the recursive traversal of the accessibility hierarchy.

Handle Action Events

If the custom subviews you create support actions, you must also handle the following events:

- `kEventAccessibleGetAllActionNames`
- `kEventAccessibleGetNamedActionDescription`
- `kEventAccessiblePerformNamedAction`

Your handler for the `kEventAccessibleGetAllActionNames` event must return the names of all actions your custom subview supports. Action names are defined in `AXActionConstants.h` (in the HIServices framework) and are of the form `AX`*Actionname*, such as `AXRaise` and `AXPush`. To supply these names, you create a CFString for each action name and insert them into the mutable CFArray in the event's *`kEventParamAccessibleActionNames`* parameter.

Unlike an action name, such as `AXRaise`, an action description is a human-intelligible, localizable string that names the action, such as "raise". If your custom subview supports an action, you must also supply the action description. If you're developing in Mac OS X version10.4 or later, you can use the `AXUIElementCopyActionDescription` convenience function (defined in `AXUIElement.h`) to get the current system-defined action description instead of hard-coding it. This way, your action description will always be consistent with the system's action descriptions, even if they change.

Use the action description to modify the mutable CFString in the event's *`kEventParamAccessibleActionDescription`* parameter.

An assistive application uses the `kEventAccessiblePerformNamedAction` event to tell an accessibility object to perform an action. If you provide a custom subview that performs actions, you must handle this event.

Beginning in Mac OS X version 10.3, the `kEventAccessiblePerformNamedAction` event includes a parameter that allows you to defer the action if performing it involves a call to a routine that might not return immediately. This can happen if your application performs a modal action, such as bringing up a menu. An assistive application waiting for confirmation of the performance of an action can time out if it takes too long. The assistive application can interpret this as a failure to perform the action (and give this feedback to the user), even if your application is running normally.

To avoid this situation, you can check the *kEventParamAccessibilityEventQueued* parameter. If it indicates the event has been queued, you can perform the action (including the calls to routines that might not return immediately) without causing an assistive application to time out. If it indicates the event has not been queued, you can request the event be queued and sent to you later. To request the event be queued, return `eventDeferAccessibilityEventErr` from your event handler.

Note: Be aware that in Mac OS X version 10.2, the *kEventParamAccessibilityEventQueued* parameter does not exist in the `kEventAccessiblePerformNamedAction` event. This means that this event is always directly dispatched and you cannot request that it be sent to you later. In this case, you must always perform the requested action, regardless of how long it takes.

Send Notifications

An assistive application relies on notifications from accessibility objects to tell it when, for example, the object appears or disappears. Carbon implements the notification mechanism for standard HIObject and HIView objects. If you create custom classes to implement subviews in your application, however, you must send some of these notifications yourself.

To broadcast a change to a custom user interface object, use the `AXNotificationHIObjectNotify` function (defined in `CarbonEvents.h`). The function requires the `HIObjectRef` and 64-bit identifier that identify the user interface object and the notification name string. If, for example, a custom window has moved, you pass the notification string "AXWindowMoved". The header file `AXNotificationConstants.h` (in the HIServices framework) defines the notification strings you can use.

Making a Custom Carbon Application Accessible

Many Carbon applications use custom application frameworks instead of newer technologies, such as HIToolbox. This chapter defines a custom Carbon application as one that fits at least some of the following criteria:

- It relies on a custom application framework.
- Its user interface implementation may not be object-oriented.
- It may not use Carbon events.

If some of these statements describe your application, you should read this chapter for guidelines on how to access-enable your application.

How Much Work Will This Be?

There is no doubt that it is more work to access-enable a custom application than an application that handles Carbon events and uses only HIObjects and HIViews in its user interface. However, the Carbon accessibility implementation allows the selective addition of accessibility support. This means you can add accessibility where and when you need it, without worrying about having to redesign your application.

Although some steps will change for individual applications, in general, you take the following steps to access-enable a custom Carbon application:

- Decide which user interface objects need to be accessible and define their relationships.
- Create an accessibility object for each accessible user interface object.
- Install event handlers on the new accessibility objects to handle the accessibility events.
- Make sure the new accessibility objects send the appropriate notifications to assistive applications.

The following sections in this chapter describe each of these steps in more detail. In addition, be sure your application does not use the `GetCurrentEventKeyModifiers` function. If it does, see "Key Modifiers and VoiceOver" (page 16) for information about why you should not use this function and a suggestion for how to replace it.

Define Your Application's Accessibility Hierarchy

As described in *Accessibility Overview*, your application presents itself to an assistive application as a hierarchy of accessibility objects. As assistive application traverses the hierarchy in response to a user's navigational commands and when determining keyboard and mouse focus. If a user interface object is not represented by an accessibility object in the accessibility hierarchy, it is invisible to an assistive application.

The first step in access enabling a custom Carbon application is to determine which objects should appear in the accessibility hierarchy.

In a standard Carbon application, the containment hierarchy defined by the HIObjects and HIViews automatically provides the fundamental structure of the accessibility hierarchy. An HIView, for example, is aware of all its children and Carbon automatically provides references to these children in response to events that ask for them.

Because your application does not rely on standard HIObjects and HIViews in its user interface, you have to define the accessibility hierarchy yourself. This requires some thought about how a user navigates and uses your application. It might be helpful to use the Accessibility Inspector application (available in `/Developer/Applications/Utilities/Accessibility Tools` in Mac OS X version 10.4) to examine the accessibility hierarchies of other applications.

With your application's accessibility hierarchy mapped out, you can proceed with the task of making each appropriate user interface object accessible

Create Accessibility Objects For Custom Objects

As described in "Creation of Accessibility Objects" (page 11), it is efficient to create accessibility objects as they are needed, in response to accessibility events, rather than creating all of them when your application launches (or a window opens). Whichever way you choose to create accessibility objects to represent your custom user interface objects, however, you must do some preparation first.

Because the Carbon implementation of accessibility objects is based on HIObject, you must create an HIObject wrapper for each accessible custom object in your user interface. You need to have an HIObject to represent a view, for example, so you have an object on which to install your custom accessibility event handlers. As you access-enable your application, you install the accessibility event handlers that supply values and information specific to your application.

To create an HIObject (or HIView) wrapper for a custom user interface object, you use the `HIObjectCreate` function.

Note that you need to register your wrapper subclass in advance with the `HIObjectRegisterSubclass` function.

Instead of creating an HIObject wrapper for each subcomponent of your complex object, create a wrapper for the top-level object and define a unique 64-bit identifier for each accessible subcomponent. To follow the Carbon convention, the identifier of the top-level object should be `0`.

Now that you have an `HIObjectRef`, you use it to create an accessibility object. To do so, use the `AXUIElementCreateWithHIObjectAndIdentifier` function, passing it the `HIObjectRef` and an identifier.

For complex objects that contain arbitrary levels of child objects, the identifier of the top-level, parent object should have the value `0`. If you need to create an accessibility object for a subcomponent, you define an identifier for that part that makes sense in your implementation.

When you create an accessibility object, you must also supply the appropriate set of attributes and values. As described in "Accessibility Object Attributes" (page 11), the most important attribute is the role attribute. The Mac OS X accessibility protocol defines a large number of roles that cover most things a user interface object can be. Assistive applications rely on the roles Mac OS X defines, so you should not create new ones.

Look through the roles defined in `AXRoleConstants.h` to find the role that most closely describes what you custom object does. If your custom object behaves like a button, for example, you should define its role attribute to be `AXButton`.

If you provide the value of the role attribute, you also have to provide the value of the role description attribute. Carbon provides the `HICopyAccessibilityRoleDescription` function that allows you to get the Apple-defined role description for a specific role so you don't have to create one yourself. Using this function ensures that your application will always be up to date, even if Apple changes a role's description.

Handle Accessibility Carbon Events

After you've created accessibility objects to represent all your accessible user interface objects, install your custom accessibility event handlers. These handlers are just like the handlers discussed in "Install Custom Event Handlers" (page 26). The only difference is in the specific events you must handle.

In general, you will need to handle all the events you handle for a semistandard Carbon application, in addition to any required by your custom user interface elements.

Send Notifications

As with the accessibility Carbon events, you handle notifications in you custom application just as you handle them in a semistandard application. The difference is in which notifications you must send. For completely custom objects, you cannot depend on Carbon to send notifications for you, so you must send them yourself. This includes the fundamental creation and destruction notifications, in addition to the notifications of object movement and value change.

See "Send Notifications" (page 28) for more information on how to send notifications.

Document Revision History

This table describes the changes to *Accessibility Programming Guidelines for Carbon*.

Date	Notes
2007-02-08	Added recommended usage of the linked UI element attribute for search fields and results.
2006-02-07	Fixed typos and clarified the steps for associating a title with a set of user interface objects.
2005-04-29	Updated with extensive rewrites and additions. Changed title from "Making Carbon Applications Accessible to Users With Disabilities."

Document Revision History