



Assistive Application Programming Guide For Mac OS X

For the PFAssistive and PFEventTaps Frameworks

Copyright © 2010 Bill Cheeseman. Used by permission. All rights reserved.

PFIDDLessoFT, PFiddle Software, pfiddle, pfiddles, and the PFIDDLessoFT logo
are trademarks of PreForm Assistive Technologies, LLC.

Table of Contents

Introduction to Assistive Application Programming Guide	1
The PFiddlesoft™ PFAssistive and PFEventTaps Frameworks	1
Related Documentation	2
Licensing Terms	2
Free for Personal Use and for Distribution and Use With Free Products	3
One-Time Licensing Fee for Distribution and Use With Paid Products	3
DISCLAIMERS	3
How to Use the PFAssistive Framework	5
Discovering UI Elements	6
1. Reading the Screen	6
2. Browsing the Accessibility Hierarchy	8
3. Observing Notifications	9
Reading a UI Element's Attributes	11
1. Reading Simple Attributes	11
2. Reading Unknown Attributes Using Key-Value Coding	12
3. Reading Parameterized Attributes	13
4. Navigating the Accessibility Hierarchy	16

Controlling an Application	17
1. Setting a UI Element's Attributes	18
2. Performing Actions on a UI Element	20
3. Sending Keystrokes to an Application	21
Miscellaneous	21
How to Use the PFEventTaps Framework	22
Monitoring Events Using an Event Tap	23
Filtering, Modifying, Blocking, and Responding to Events	27
Posting Synthetic Events	29
Reading the State of an Event Source	31
How to Enable Access for Assistive Devices	33

Introduction to Assistive Application Programming Guide

This *Programming Guide* explains how to write an assistive application for Macintosh computer users with disabilities using the PFiddlesoft PFAssistive and PFEEventTaps Frameworks. The PFiddlesoft Frameworks are written in Objective-C and designed for use with Cocoa applications. They support and enhance Apple's Accessibility and Quartz Event Taps APIs, enabling Cocoa developers to use familiar programming techniques to create assistive applications without having to master the technicalities of Apple's procedural C Accessibility, Core Foundation, and Core Graphics APIs.

Apple's Accessibility technology grew out of Section 508 of the Workforce Investment Act of 1998 and its requirements regarding access to electronic and information technology for persons with disabilities. Compliance with Section 508 is a prerequisite for sale of computer and other products to the federal government and to many state agencies and educational institutions. The Accessibility API is designed for use both by developers incorporating its features into their own accessible applications and by developers of assistive devices and applications for users with disabilities. The Event Taps API is also a Section 508 enabling technology.

Because Accessibility is built into every standard Mac OS X User Interface element, whether written using the Cocoa or Carbon frameworks, it is capable of much broader uses. Software testing tools, network administration tools, troubleshooting tools, plug-ins for applications that don't have a plug-in architecture, and remote control applications are only some of the possibilities.

The PFiddlesoft™ PFAssistive and PFEEventTaps Frameworks

The PFAssistive Framework was created in 2003 as the engine driving PFiddlesoft's highly regarded UI Browser utility for developers and for users of Apple's AppleScript GUI Scripting technology. The PFEEventTaps Framework was added in 2007 as the engine underlying PFiddlesoft's Event Taps Testbench utility for developers. Both frameworks have been revised and updated over a period of years, and they have demonstrated their power and reliability in PFiddlesoft's commercial and free developer utilities. PreForm Assistive Technologies, LLC is making the PFiddlesoft Frameworks available to all Macintosh developers. Together, the PFiddlesoft Frameworks bring to Cocoa developers the full range of Accessibility capabilities needed to write assistive applications and other software that explores, manipulates, and monitors the User Interface elements and user inputs of most Mac OS X applications.

The current versions of the PFiddlesoft Frameworks require Mac OS X 10.5 Leopard or newer, and they support all Accessibility and Event Taps features introduced by Apple through Mac OS X 10.6 Snow Leopard. They are universal binaries supporting clients that run natively on PowerPC or Intel processors using 32-bit or 64-bit architectures with reference counted memory management. Support for garbage-collected client applications is not yet available. They are intended for installation as shared frameworks in the local /Library/Frameworks folder. They should not be embedded in

an assistive application's bundle because this will prevent the assistive application from being treated as a “trusted” application; see [How to Enable Access for Assistive Devices](#) for more information.

The two frameworks are independent of one another. Both together enable you to write a full-featured assistive application, but you may need only one or the other to accomplish more limited purposes.

Download the PFiddlesoft Frameworks at pfiddlesoft.com. PFiddlesoft's free developer utilities and free 30-day trial versions of its commercial products are also available for download there.

Related Documentation

In addition to this *Programming Guide*, consult the *PFAssistive Framework Reference* and the *PFEEventTaps Framework Reference* for detailed documentation of every public method and property made available in the frameworks. These References are embedded in the Resources folders of the frameworks, and they are available for download at pfiddlesoft.com. Sample code in the form of a simple screen reader is provided with the framework.

Apple's Accessibility API is a set of C header files located in the HIServices subframework of the Mac OS X ApplicationServices framework, in /System/Library/Frameworks. The API was introduced in Mac OS X 10.2.0 Jaguar and is installed by default on every Macintosh computer running Mac OS X 10.2 or newer. See Apple's *Accessibility (ApplicationServices/HIServices) Reference* and *Accessibility Roles and Attributes Reference* for documentation of the C API, and also consult the comments in the header files. Sample code for building an assistive application using the C API is available in Apple's UIElementInspector 1.3 sample.

Apple's documentation for making applications accessible (sometimes called “access enabling” or “accessorizing” an application) is also useful in understanding how to write assistive applications that take advantage of any application's accessibility features. See Apple's *Accessibility Programming Guidelines for Cocoa*, *NSAccessibility Protocol Reference*, *Accessibility Programming Guidelines for Carbon*, and *Carbon Accessibility Reference*. Also consult Apple's *Accessibility Overview* and its *Getting Started with Accessibility* document.

Apple's Event Taps API is a set of C header files located in the CoreGraphics subframework of the Mac OS X ApplicationServices framework, in /System/Library/Frameworks. The API was introduced in Mac OS X 10.4.0 Tiger and is installed by default on every Macintosh computer running Mac OS X 10.4 or newer. See Apple's *Quartz Event Services Reference* for documentation of the C API, and also consult the comments in the header files.

You will find it easier to follow and understand this *Programming Guide* and the companion *PFAssistive Framework Reference* and *PFEEventTaps Framework Reference* if you download the free 30-day trial version of PFiddlesoft's UI Browser application and the free PFiddlesoft Event Taps Testbench utility. These developer utilities demonstrate the features of Apple's Accessibility and Quartz Event Taps APIs. Download them, as well as the PFAssistive and PFEEventTaps Frameworks and related documentation, at pfiddlesoft.com.

Licensing Terms¹

The PFiddlesoft Frameworks are copyrighted software.

¹ This is a summary of the PreForm Assistive Technology, LLC licenses. For the legally binding terms consult the licenses themselves. The licenses are embedded in the frameworks' bundles, and they are also available for download at pfiddlesoft.com.

Free for Personal Use and for Distribution and Use With Free Products

The PFiddlesoft Frameworks may be licensed free of charge for personal use, including use during development of any client application or other software. They may also be licensed free of charge for distribution and use with any client application or other software that you distribute to the public free of charge (including freeware as well as free beta or trial versions of a product for which you intend to request or require payment in the future). You are required only to give notice to PreForm Assistive Technologies, LLC, to provide attribution to PreForm Assistive Technologies, LLC in your client application or other software, and to include the copyright notice and license in your client application or other software.

One-Time Licensing Fee for Distribution and Use With Paid Products

If you distribute the PFiddlesoft Frameworks with or in a client application or other software product for which you request or require payment, or if you distribute a client application or other software product that includes or uses the PFiddlesoft Frameworks for which you request or require payment, such as donationware, shareware, and commercial applications, or for internal use within a for-profit organization, you must within thirty days of initial distribution of your product pay PreForm Assistive Technologies, LLC a flat one-time license fee of \$250 U.S. for each framework that you distribute or use, regardless of the number of units of your product you distribute or use. This fee covers all present and future versions of your product, but any separate and distinct product requires you to pay PreForm Assistive Technologies, LLC an additional licensing fee of \$250 U.S. for each framework that you distribute or use, as described above.

An executed license is required both for free distribution or use, and for distribution or use subject to a flat one-time license fee with a product for which you request or require payment. Download the [PFAssistive Framework distribution license](#) or the [PFEventTaps Framework distribution license](#) or both of them, depending on which of the PFiddlesoft Frameworks you distribute or use. Then print the licenses in duplicate, fill in the blanks, sign them, and mail them to:

PreForm Assistive Technologies, LLC
P.O. Box 326
Quechee, VT 05059-0326

DIFFERENT TERMS APPLY TO LARGE OR ESTABLISHED COMMERCIAL SOFTWARE DEVELOPERS. The source code is available for an additional fee. Contact us at sales@pfiddlesoft.com for details.

DISCLAIMERS

The PFiddlesoft Frameworks are provided on an "AS IS" basis. The following disclaimers apply to each of the frameworks:

PREFORM ASSISTIVE TECHNOLOGIES, LLC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE FRAMEWORK OR ITS USE AND OPERATION ALONE OR IN COMBINATION WITH OTHER PRODUCTS. THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

IN NO EVENT SHALL PREFORM ASSISTIVE TECHNOLOGIES, LLC BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF

SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) ARISING IN ANY WAY OUT OF THE USE, REPRODUCTION, MODIFICATION AND/OR DISTRIBUTION OF THE FRAMEWORK, HOWEVER CAUSED AND WHETHER UNDER THEORY OF CONTRACT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY OR OTHERWISE, EVEN IF PREFORM ASSISTIVE TECHNOLOGIES, LLC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. YOUR REMEDY FOR ANY DEFECT OR FAULT IN THE FRAMEWORK IS LIMITED TO REFUND OF THE LICENSE FEE YOU PAID.

How to Use the PFAssistive Framework

Apple's Accessibility API implements the concept of a *UI element*, an object that represents a user interface element on the screen in any running application, such as a menu, a window, or a button, or the application itself. The Accessibility API also implements the concept of an *observer*, an object that registers to observe a UI element that issues Accessibility notifications when changes occur.

The PFAssistive Framework implements these same concepts in its PFUIElement, PFApplicationUIElement, and PFObserver classes, each of which instantiates and encapsulates an associated Accessibility API object and makes its capabilities available to an assistive application using standard Objective-C and Cocoa techniques. For example, an assistive application using the PFAssistive Framework can implement optional delegate methods declared in the framework's PFUIElementDelegate and PFObserverDelegate formal protocols to respond to Accessibility notifications.

In this chapter, you learn how to discover UI elements, how to read UI element attributes, and how to control UI elements and, through them, the target application, all in the interest of supporting assistive applications that enable a user with disabilities to use the computer to perform the same tasks that any user can perform with the graphical user interface. An assistive application typically does this by performing these tasks:

- An assistive application discovers individual UI elements in the target application in one of three ways: by locating an element on the screen, for example, the element currently under the mouse or the element that is the target application's frontmost window; by receiving a notification from an element that something about it has just changed; or by navigating the element hierarchy from a known starting point. Read the [Discovering UI Elements](#) section for details.
- When an assistive application discovers a UI element of potential interest, it ascertains the element's identity and nature by reading its attributes, such as its role, its title, and its position and size on the screen. Read the [Reading a UI Element's Attributes](#) section for details.
- Once it identifies and understands a UI element, an assistive application manipulates or controls the element and, through it, the target application, at the user's direction. There are three ways to do this, depending on the nature of the element: by setting the element's value or other attribute; by performing an action on the element; or by sending a keystroke to the target application while the element has keyboard focus (or by sending a key combination that the application recognizes as a keyboard shortcut). Read the [Controlling an Application](#) section for details.

By performing these tasks—discovering, reading, and controlling a UI element—an assistive application enables a user with disabilities to do everything that a user without disabilities can do with the target application. This parity of treatment defines and delimits the Accessibility API. Apple rigorously enforces the notion that the Accessibility API should enable a user with disabilities to do everything that any user can do with a target application, and no more. What the API cannot

do is as important as what it can do. For example, the Accessibility API cannot read or control UI elements in a window that is offscreen, even if the window still exists in memory and the Cocoa frameworks can see it.

This design principle explains a key feature of the Accessibility API, namely, the hierarchy of UI elements that an assistive application navigates to find any element's parent and children. In principle, the Accessibility hierarchy includes only those elements that a user can read and control in the graphical user interface of the target application. Although the Cocoa view hierarchy includes many invisible views that contain other views for purposes of programmatic control, the Accessibility hierarchy ignores them because they play no direct role in a user's work with the target application.

In the first section of this chapter, you learn how to discover a UI element, including the application UI element and the special system-wide UI element. This involves using the `PFUIElement` class and its `PFApplicationUIElement` subclass to read the screen or to browse an application's Accessibility hierarchy, or using the `PFObserver` class to register for and observe notifications from an application or its individual UI elements. In the second section, you learn how to read a UI element's attributes. In the last section, you learn how to control an application by setting those attributes of its UI element that are settable, by performing actions that are recognized by some of its UI elements, and by sending keystrokes to it.

Discovering UI Elements

There are three primary ways in which an assistive application discovers UI elements:

1. A screen reader is the most common kind of assistive application. It ascertains the identify of the UI element currently under the mouse or at a specified position on the screen, either in the active application or in an application that is running but is not currently the frontmost application. Examples are Apple's VoiceOver application, its Accessibility Inspector, and the Screen Reader in PFiddlesoft's UI Browser. Read the [Reading the Screen](#) subsection for details.
2. A browser is another kind of assistive application. The user specifies a target application and then navigates that application's Accessibility hierarchy until it finds a UI element of interest. Examples are Apple's System Events application, which supports GUI Scripting for AppleScript using the Accessibility API, and the main browser view in UI Browser. Read the [Browsing the Accessibility Hierarchy](#) subsection for details.
3. An observer is a third kind of assistive application. It registers to monitor specified kinds of activity in a target application or a particular UI element, and it responds when it receives a notification of interest. UI Browser is an example. Read the [Observing Notifications](#) subsection for details.

Some assistive applications, such as UI Browser, implement all three modes of operation.

1. Reading the Screen

A screen reader discovers UI elements by determining what UI element is located at a specified point on the screen or screens attached to the computer, typically the point where the mouse pointer is located. To implement this feature in your assistive application, use the `+elementAtPoint:withDelegate:error: class` method of the `PFUIElement` class or the `-elementAtPoint:` method of the `PFApplicationUIElement` class, both declared in `PFUIElement.h`.

Use the `PFUIElement` class method when an assistive application needs to know what UI element is visible at the specified location without regard to which running application owns it. Use the `PFApplicationUIElement` instance method when it needs to know what UI element belonging to a specific application is at the specified location, even if the element is currently obscured by a UI element belonging to another application.

Both of these methods take an `NSPoint` structure specifying a point on the screen as an argument. The Accessibility API uses the Quartz 2D device space coordinate system in which the origin is at the top-left corner of the primary screen, the screen on which the menu bar appears.² Positive movement flows down and to the right; negative movement, up and to the left. From the Cocoa perspective, device space is flipped.

A screen reader typically focuses on the current mouse pointer. In Cocoa, the location of the mouse pointer can be obtained at any time in bottom-left relative screen coordinates, regardless of the current event or pending events, using `NSEvent's +mouseLocation` class method. You must convert it to the Accessibility API's top-left relative screen coordinates before passing it to either of these methods. To avoid having to convert the coordinates, use Carbon's `HIGetMousePosition()` function, which returns top-left relative screen coordinates; it is available in 64-bit applications as well as 32-bit applications.

An assistive application may need to read the screen continuously as the user moves the mouse, or it may only need to read the screen at user direction, for example, in response to a keyboard shortcut, a global hot key, or an AppleScript command. To read the screen continuously, an application would typically use a repeating timer with a resolution that is tight enough to capture everything the mouse moves over, perhaps 2 or 3 times per second. To read the screen at user direction, simply call `+elementAtPoint:withDelegate:error:` or `-elementAtPoint:` in an action method or other method that is called on command.

Here is a simplified version of the `-updateScreenReader` method in UI Browser's screen reader window controller:

```
- (void)updateScreenReader {
    HIPoint location;
    HIGetMousePosition(kHICoordSpaceScreenPixel, NULL, &location);
    NSPoint point = NSMakePoint(location.x, location.y);
    PFUIElement *element =
        [PFUIElement elementAtPoint:point withDelegate:nil error:NULL]];
    // ...
}
```

UI Browser's `-updateScreenReader` method is called from a repeating timer, which is set up in the window controller's `-awakeFromNib` method like this:

```
[NSTimer scheduledTimerWithTimeInterval:0.4 target:self
    selector:@selector(updateScreenReader) userInfo:nil repeats:YES];
```

The `+elementAtPoint:delegate:error:` method takes two other arguments, `delegate` and `error`. They may be `nil` or `NULL`, respectively, as shown here. Pass an object in the `delegate` argument and implement either or both of the `PFUIElementDelegate` formal protocol methods `-PFUIElementWasDestroyed:` and `-PFUIElementReportError:` if you wish to take advantage of those capabilities. The `error` argument is an indirect reference to a standard `NSError` object. See the *PFUIElement Class Reference* for more information.

² The primary screen may differ from the screen returned by `NSScreen's -mainScreen` method, which returns the screen with the active window.

Both `+elementAtPoint:withDelegate:error:` and `-elementAtPoint:` create and return an autoreleased `PFUIElement` object representing the UI element at the specified location. An assistive application can read its attributes, navigate the Accessibility hierarchy from it using its parent and children attributes, register to observe changes to it, and control it using the other features of the PFAssistive Framework.

2. Browsing the Accessibility Hierarchy

A browser discovers UI elements by navigating the Accessibility hierarchy from a known starting point, typically the *root application element* of a specified application or the *system-wide element* that is aware of frontmost running application. To do this in your assistive application, start by creating an application or system-wide UI element using one of the initializers declared in the `PFApplicationUIElement` class, `-initWithPath:delegate:`, `-initWithPid:delegate:`, and `-initWithSystemWideWithDelegate:`, all declared in `PFUIElement.h`.

The easiest way to create an application UI element is to use the path to the application file. To access `TextEdit`, for example, use any available Cocoa technique to get the path to `TextEdit`'s application file, such as spelling it out like this:

```
NSString *appPath = @"/Applications/TextEdit.app";
```

or getting it from the bundle identifier like this:

```
NSString *appPath = [[NSWorkspace sharedWorkspace]
    absolutePathForAppBundleWithIdentifier:@"com.apple.TextEdit"];
```

Then create and initialize the application UI element like this:

```
PFApplicationUIElement *appElement =
    [[PFApplicationUIElement alloc] initWithPath:appPath delegate:nil];
```

The designated initializer for the `PFApplicationUIElement` class is `-initWithPid:delegate:`. Use the designated initializer when an application has easy access to the BSD Unix process identifier number (PID) of the target application. Every `PFUIElement` object makes its application PID available in its `-pid` method, so the application can use the designated initializer to create a `PFApplicationUIElement` object for the target application whenever a `PFUIElement` object is available, for example, after reading the screen. Alternatively, simply call the `PFUIElement` object's `-applicationElement` method.

The Accessibility API recognizes a special UI element known as the system-wide element. Use it when an application needs to get a `PFApplicationUIElement` object representing the application that is currently active, using `PFUIElement`'s `AXFocusedApplication` property.

The delegate argument to the three `PFApplicationUIElement` initializers plays the same role that it plays in `+elementAtPoint:withDelegate:error:`, discussed above. Set it to `nil` if the `PFUIElement` delegate methods are not needed.

The first two initializers create and return an autoreleased `PFApplicationUIElement` object representing a running application. The third creates and returns an autoreleased `PFApplicationUIElement` object representing the system, from which the active or frontmost running application can be determined by reading its `AXFocusedApplication` attribute.

An assistive application can read a target application element's attributes, register to observe changes to it, and control it using the other features of the PFAssistive Framework. Most importantly for purposes of browsing, it can navigate the Accessibility hierarchy by using the **AXParent** and **AXChildren** attributes to get every element in the hierarchy. The root application UI element's **AXChildren** property is the starting point. For more about reading Accessibility attributes to navigate the hierarchy using PFUIElement properties, consult the [Navigating the Accessibility Hierarchy](#) subsection of the [Reading a UI Element's Attributes](#) section.

3. Observing Notifications

An observer discovers UI elements by registering to be notified when a target application's UI elements are changed by user activity or by other means such as an AppleScript script or another assistive application. When the user of a target application types into a text field, adjusts a slider, moves or resizes a window, or selects a menu item, for example, an observer that has registered to observe such activity receives a notification. The notification includes a reference to the affected UI element, and the observer can then use it to respond.

To observe Accessibility notifications in your assistive application, create a PFObserver object using one of the factory convenience methods declared in PFObserver.h, or create a PFObserver object and initialize it with one of the declared initializers. Once a PFObserver has been created, register it to observe a particular notification using the **-registerForNotification:fromElement:contextInfo:** method.

A PFObserver has a choice of two techniques to watch for and respond to Accessibility notifications: a delegate method and a callback method. The delegate method is easier to use because it provides full observer management, and it suffices for most purposes. The callback method allows for more complex behavior when circumstances require it, but the assistive application must take responsibility for managing the observer. The choice is made when the PFObserver is created.

To create a PFObserver that uses a delegate method, call the **+observerWithName:** factory method, passing in the name (not the path) of the target application. Then, when the assistive application is ready to begin observing the target application, take two additional steps: set the delegate, and register to observe a specified notification from a specified UI element. The delegate must declare that it conforms to the PFObserverDelegate formal protocol, which is declared in PFObserver.h. Set the delegate by sending PFObserver's **-setDelegate:** message.

To create a PFObserver that uses a callback method instead of a delegate method, call the **+observerWithName:notificationDelegate:callback:** factory method, passing in the name (not the path) of the target application, along with an object to serve as the notification delegate and the selector for the callback method that the notification delegate implements.

As an alternative to using the target application's name, call factory methods that use the target application's BSD Unix process identification number (PID) to create a PFObserver, **+observerWithPid:** and **+observerWithPid:notificationDelegate:callback:**.

All four of the factory methods have corresponding initializers for use when factory methods are inappropriate. Two of the initializers, **-initWithPath:** and **-initWithPath:notificationDelegate:callbackSelector:**, take the target application's full path (not its name). The designated initializers are **-initWithPid:** and **-initWithPid:notificationDelegate:callbackSelector:**.

Registration using the **-registerForNotification:fromElement:contextInfo:** method requires two arguments, **notification**, a string identifying a valid Accessibility API notification, and **element**, the PFUIElement

that is to be observed. The `contextInfo` argument is optional; it is provided in case an assistive application needs to pass information to the observer for use when a notification is received. The element to be observed can be a `PFAApplicationUIElement`. This is often desirable, because an observed application element issues notifications whenever the specified notification is issued by any UI element in the target application. For example, if the target application is observed for `kAXWindowMovedNotification`, it issues a notification when any window in the application is moved, and the notification identifies the affected window. If a specific window is observed, it issues a notification only when that window is moved. In that case, the observed element and the affected element are one and the same element.

One `PFObserver` can be registered to observe many notifications, or an assistive application can create multiple `PFObservers` and register each of them to observe different notifications. For example, UI Browser creates a single `PFObserver` object and registers it to observe many notifications selected by the user from a table in UI Browser's Notifications drawer. It uses a callback method to respond to notifications. This code, from the repeat block governing a single row in the table, calls a number of internal UI Browser methods that are not explained here, but the pattern should be clear:

```
// Lazily create observer if not yet created.
if (![self notificationObserver]) {
    [self setNotificationObserver:
        [PFObserver observerWithPid:[self currentElement] pid]
        notificationDelegate:self callbackSelector:
            @selector(observer:notification:element:contextInfo:)];
}

// Register observer for notifications and element.
NSIndexSet *indexSet = [table selectedRowIndexes];
NSMutableInteger indexBuffer[[indexSet count]];
NSMutableInteger idx;
for (idx = 0; idx < [indexSet getIndexes:indexBuffer
    maxCount:[indexSet count] inIndexRange:NULL]; idx++) {
    [[self notificationObserver] registerForNotification:
        [(NSDictionary *)[combinedNotifications objectAtIndex:indexBuffer[idx]]
        objectForKey:@"notification"] fromElement:[self currentElement]
        contextInfo:(void *)[self currentElement] elementInfo]];
}
```

The payoff comes when notifications are issued in response to user-initiated changes in the target application's graphical user interface. In the case of a delegate-based observer, the delegate method declared in the `PFObserverDelegate` protocol, `-applicationWithIdentifier:atPath:didPostAccessibilityNotification:fromObservedUIElement:forAffectedUIElement:`, is called. In the case of a callback-based observer, the callback method declared in the notification delegate is called. It must have this signature:

```
- (void)observer:(PFObserver *)observer notification:(NSString *)notification
    element:(PFUIElement *)element contextInfo:(void *)contextInfo
```

The delegate method is provided with several useful items of information: the bundle identifier and path of the target application, the notification string, the observed PFUIElement and the affected PFUIElement. The callback method is provided with somewhat different information, but all needed items can be determined from the PFObserver object that is passed in. The key item in both cases is the affected UI element. An assistive application can read its attributes, navigate the Accessibility hierarchy from it using its parent and children attributes, register to observe other changes to it, and control it using the other features of the PFAssistive Framework.

In addition to PFObserver delegate and callback methods, an assistive application can implement optional delegate methods declared in the PFUIElement class or register to receive corresponding notifications whenever a UI element is destroyed in the user interface, or to learn of Accessibility errors. See the UIElementDelegate formal protocol declared in PFUIElement.h.

Reading a UI Element's Attributes

Some assistive applications are passive. They discover the target application's UI elements and read the values of their attributes, but they do not enable the user to control the target application. Even active assistive applications typically must read the attributes of the target application's elements.

There are four important kinds of operations involved in reading attributes of a UI element:

1. Reading any of dozens of simple attributes by name to obtain information about the current state of a UI element. Read the [Reading Simple Attributes](#) subsection for details.
2. Reading any of the simple attributes by using Key-Value Coding (KVC) when their names are not known until runtime. This operation is very powerful, because in many situations an assistive application can get a list of relevant attribute names and iterate over them to get their values. In addition, this operation allows an assistive application to read new attributes introduced in future versions of Mac OS X and custom attributes used by third-party applications. Read the [Reading Unknown Attributes Using Key-Value Coding](#) subsection for details.
3. Reading several so-called parameterized attributes to obtain the value of some part of a complex UI element given a variable parameter such as the position of the mouse on the screen. For example, an assistive application can get a character or line of text under the mouse in a text area or a cell at the intersection of a given column and row in a cell-based table. Read the [Reading Parameterized Attributes](#) subsection for details.
4. Reading the AXParent and AXChildren attributes of any UI element to navigate the accessibility hierarchy. Read the [Navigating the Accessibility Hierarchy](#) subsection for details.

1. Reading Simple Attributes

An assistive application normally reads the values of a UI element's Accessibility attributes using the properties declared in the Attributes category on the PFUIElement class. The Attributes category declares several dozen properties, each corresponding to an Accessibility API attribute. The value of an Attributes property is always a Cocoa object. Each Attributes property is named for the NSString used to identify the attribute in the Accessibility API, such as **AXRole** for the attribute named "AXRole" and **AXTitle** for the attribute named "AXTitle."

Depending on the attribute, the value of an Attributes property of the receiving UI element may be another PFUIElement object or an NSString, NSNumber, NSValue, or other object. For example, the value of the **AXParent** property is a PFUIElement object representing the receiving PFUIElement's parent element in the Accessibility hierarchy. Some properties return an NSArray object containing multiple PFUIElements, NSStrings, or other objects. For example, the **AXChildren** property returns an array of PFUIElement objects representing the receiving PFUIElement's children

elements in the hierarchy. For `NSNumber` and `NSValue` attribute values, an assistive application must use standard Cocoa methods to extract the value, such as `-boolValue`, `-intValue`, `-pointValue`, and `-rangeValue`. For example, the `AXSize` property returns an `NSValue` object representing the height and width of the receiving `PFUIElement` on the screen; use `NSValue`'s `-sizeValue` method to extract its `NSSize` structure. Consult the *PFAssistive Framework Reference* for information about each property, including its type, the type of its contents if it is an `NSArray`, and the method to use to extract its value if it is an `NSNumber` or `NSValue`.

The `PFUIElement` class includes several utility methods that help an assistive application to read UI element attributes. The `-exists` method returns `YES` if the receiving UI element still exists in the target application's graphical user interface; for example, if a window has not been closed by the user. It is prudent to test whether an element still exists before using it, because objects representing destroyed elements may be `nil` or may have been recycled to refer to other elements, leading to unexpected behavior. The `-existsAttribute:` method returns `YES` if the receiving element exists and the name of the attribute passed in the `attribute` argument identifies an attribute that is supported by the receiving `PFUIElement`. The `-existsValueForAttribute:` method returns `YES` if the receiving element exists, it supports the attribute, and the attribute returns a value. This is important because Cocoa applications often indicate that they support an attribute but do not in fact return a value for it. The `-typeForAttribute:` method returns an `NSString` identifying the type of an attribute's value, such as "array," "Boolean," "range," "rect," or "UIElement," suitable for use as the identifier of a table column, for determining which `NSNumber` or `NSValue` method to use to extract a structure, and in other situations where an attribute's value must be processed differently depending on its type.

This example from UI Browser reports either the plain English description or the technical name (beginning with "AX") of each attribute listed in UI Browser's Attributes drawer, depending on a user preference setting.

```
NSString *description;
if ([[NSUserDefaults standardUserDefaults]
    integerForKey:TERMINOLOGY_STYLE_DEFAULTS_KEY] == 0) {
    description = [element AXRoleDescription];
} else {
    description = [element AXRole];
}
```

Many methods in the `PFUIElement` class return `PFUIElement` or `PFApplicationUIElement` objects. Think of these methods as factories that can be called upon to churn out these objects as often as needed, whenever needed, even if they are used only for a moment. In addition, an assistive application can create a temporary `PFUIElement` object at any time, for example, an object representing a known UI element in order to test it for equality with a cached element. The application can allocate as many UI elements as desired in this way, as often as desired, and initialize each of them with `-initWithElementRef:delegate:`. Typically, in this situation, it gets the `CGElementRef` argument from an existing or newly-created element by sending it an `-elementRef` message. Separate `PFUIElement` or `PFApplicationUIElement` objects representing the same UI element in the running application are interchangeable. Whether they represent the same element can be tested using `-isEqual:` or `-isEqualtoElement:`.

2. Reading Unknown Attributes Using Key-Value Coding

There are many circumstances in which an assistive application does not know the names of the attributes it will read until it is running. This frequently happens, for example, when reading all of the attributes of a UI element currently under

the mouse and calling PFUIElement's `-attributes` method to get an array of the element's attribute names. It may also happen when targeting an application that implements custom UI elements with custom attributes.

The PFAssistive Framework includes two methods based on Key-Value Coding (KVC) for these situations, `-valueForAttribute:` and `-valuesForAttributes:`. They take advantage of the fact that the name of every simple Attributes property is the same as the NSString used in the Accessibility API to identify it. The `-valueForAttribute:` method takes as its argument an NSString containing the name of an attribute, and the `-valuesForAttributes:` method takes an array of NSStrings containing the names of several attributes. The attributes passed to these two methods, for example, the attributes returned by the `-attributes` method, may be unknown to your assistive application or to the Accessibility API. Nevertheless, these KVC-based methods return their values. Use the `-exists`, `-existsAttribute:`, `-existsValueForAttribute:`, and `-typeForAttribute:` methods to help understand and process the values they return.

An assistive application based on the PFAssistive Framework continues to work even under new versions of Mac OS X that introduce new attributes and with new target applications that use custom UI elements with custom attributes.

This example from UI Browser is the action method for the Find in Browser button in UI Browser's Attributes drawer:

```
- (IBAction)elementFindButtonAction:(id)sender {
    if ([self isTargetAccessible]) {
        if ([self currentElement] exists) {
            NSTableView *table = [self attributeTable];
            NSString *selectedAttribute = [(NSDictionary *)[[self cachedAttributeArray]
                objectAtIndex:[table selectedRow]] objectForKey:@"attribute"];
            PFUIElement *selectedElement = [[self currentElement]
                valueForAttribute:selectedAttribute];
            [self displayElementInCurrentApplication:selectedElement];
        } else {
            NSBeep();
        }
    }
}
```

3. Reading Parameterized Attributes

The Accessibility API recognizes several attributes of a special kind known as *parameterized attributes*. These are attributes that have a different value depending on the value of an argument passed to them. For example, parameterized attributes recognized by a text area UI element return the character located at a given point in the element's coordinate system, the NSString constituting a line of text in the view given the line number, and the bounds of a text passage in the view given the range of characters. Similarly, a parameterized attribute recognized by a cell-based table in Apple's Numbers application returns the cell located at a given row and column intersection, and parameterized attributes recognized by a layout area in Numbers convert between locations in screen coordinates and locations in scaled layout area coordinates.

The Attributes category of the PFUIElement class declares methods for all of these parameterized attributes, such as `-AXLineForIndex:`, `-AXCellForColumnAndRow:`, and `-AXLayoutPointForScreenPoint:`. An assistive

application can use them in a variety of ways. For example, a screen reader can get the location of the mouse pointer on the screen, and then use the `-AXRangeForPosition:` parameterized attribute method to obtain the range of the Unicode glyph at that location. The `location` field of the `NSRange` structure returned by sending the `-rangeValue` message to the returned `NSValue` object is the index of the first character in the glyph in the Cocoa text storage object under the mouse, and the `length` field is the distance to the character at the beginning of the next glyph, bearing in mind that Unicode glyphs may be composed of several Unicode characters. These values apply even if the text storage object spans several text containers such as multiple columns or pages. An assistive application uses the character's range as a springboard into the other parameterized attribute properties, such as `-AXLineForIndex:`, `-AXRangeForLine:`, and `-AXAttributedStringForRange:`.

To help an assistive application distinguish between simple and parameterized attributes, the `PFUIElement` class supplements its `-attributes` method with two similar methods that return lists of attributes, `-nonParameterizedAttributes` and `-parameterizedAttributes`. In addition, the kind of a particular attribute can be determined with the `-isParameterizedAttribute:` method.

The example from UI Browser on the next page gets several parameterized values in three steps:

```

- (void)recordParametersForParameterizedElement:(PFUIElement *)element
    atLocation:(NSPoint)location {
    // Get the string, range and bounds of the line of text under the mouse.
    NSValue *mousePosition = [NSValue valueWithPoint:location];

    // Get the primary parameterized value: (1) Get the index of the first character
    composing the glyph under the mouse.
    if ([element isRole:NSAccessibilityTextAreaRole]
        || [element isRole:NSAccessibilityTextFieldRole]) {
        NSValue *characterRangeForPosition =
            [element AXRangeForPosition:mousePosition];
        NSNumber *characterIndex =
            [NSNumber numberWithInt:[characterRangeForPosition rangeValue].location];

        // Get the secondary parameterized values: (2) use the character index to get
        the range of characters composing the glyph under the mouse (this should be the same
        as characterRangeForPosition); (3) use the character index to obtain the line index of
        the line under the mouse; (4) use the line index to obtain the range of characters
        composing the line under the mouse; and (5) use the character index to obtain the
        range of characters composing the style run under the mouse.
        NSValue *characterRangeForGlyph = [element AXRangeForIndex:characterIndex];
        NSNumber *lineIndex = [element AXLineForIndex:characterIndex];
        NSValue *characterRangeForLine = [element AXRangeForLine:lineIndex];
        NSValue *characterRangeForStyleRun =
            [element AXStyleRangeForIndex:characterIndex];

        // Get the tertiary parameterized values: (6) Get the string for the line; (7)
        Get the RTF data for the line; (8) Get the bounds of the line. UI Browser's use of the
        line located under the mouse as the base for calculating tertiary values is arbitrary;
        any other range could have been used as the base (such as characterRangeForStyleRun or
        characterRangeForGlyph).
        NSString *stringForLine = [element AXStringForRange:characterRangeForLine];
        NSData *RTFDataForLine = [element AXRTFForRange:characterRangeForLine];
        NSAttributedString *attributedStringForLine =
            [element AXAttributedStringForRange:characterRangeForLine];
        NSValue *boundsForLine = [element AXBoundsForRange:characterRangeForLine];

        // Store all parameter values in a dictionary.
        // ....
    }

```

4. Navigating the Accessibility Hierarchy

Many Accessibility attributes return UI elements. These attributes are yet another way in which an assistive application can discover UI elements in a target application, in addition to the three described in the [Discovering UI Elements](#) section. For example, the **AXCloseButton** property returns the PFUIElement object representing the close button in the title bar of the receiving window UI element, and the **AXSelectedRows** property returns an array of PFUIElement objects representing the rows in a table or outline.

An assistive application navigates or browses the target application's Accessibility hierarchy by using certain of these attributes after discovering the starting point by reading the screen, getting the application element, or receiving a notification. Specifically, the **AXParent** property enables an assistive application to navigate “up” the several levels of the hierarchy toward the root application element at the top, and the **AXChildren** property enables it to navigate “down” the hierarchy toward a leaf element visible on the screen. The root application element has no parent (its **AXParent** property is `nil`), and a leaf element has no children (its **AXChildren** property is an empty array).

There are several Accessibility attributes that allow an assistive application to skip levels when navigating to an important container in the target application's Accessibility hierarchy, such as the window, drawer, or sheet containing the UI element under the mouse. The **AXWindow** property returns the PFUIElement representing the receiving PFUIElement's containing window, ignoring any subcontainer levels that might lie between the window and the leaf element under the mouse. The **AXTopLevelUIElement** property is similar, but the PFUIElement it returns can represent something other than a window, such as a drawer or a sheet. The top-level container of an application dock item is the Dock application's list of dock items. To ascertain what kind of UI element the container is, get its **AXRole** property. To determine whether it is a given kind of UI element, use PFUIElement's `-isRole:` method.

The example on the next page is part of UI Browser's screen reader window controller's code to detect and report mismatches in a target application's accessibility hierarchy. An important requirement of the Accessibility API is that the hierarchy when looking “up” from a leaf element on the screen should be a mirror image of the hierarchy when looking “down” from the root application element. Otherwise, a screen reader will be unable to navigate to an element that a browser can reach, or *vice versa*. A mismatch should always be considered a bug in the target application. It exists wherever a given UI element's **AXParent** attribute omits the given UI element from its **AXChildren** attribute, and wherever a given UI element's **AXChildren** attribute contains one or more elements whose **AXParent** attribute is not the given element.

```

if (![element isRole:NSAccessibilityApplicationRole]) { // root element has no parent
    PFUIElement *browserParent = [[self elementArray] objectAtIndex:(column - 1)]
        objectAtIndex:[self elementBrowser] selectedRowInColumn:(column - 1)]];
    if (![element AXParent]) {
        [cell setValue:[NSString stringWithFormat:@"[MISMATCH-no parent] ",
            @"Warning string for no parent mismatch in target application's UI element
            hierarchy for browser"] stringByAppendingString:
            [self descriptionWithTitleAndIndexofElement:element atColumn:column]];
        NSLog(@"Mismatch browsing the children tree from the root UI element: the %@ is
            a child of %@ <%p>, but it has no parent.", [element AXRoleDescription],
            [browserParent AXRoleDescription], browserParent);
    } else if (![element AXParent] isEqualToElement:browserParent) {
        [cell setValue:[NSString stringWithFormat:@"[MISMATCH-different parent] ",
            @"Warning string for different parent mismatch in target application's UI
            element hierarchy for browser"] stringByAppendingString:
            [self descriptionWithTitleAndIndexofElement:element atColumn:column]];
        NSLog(@"Mismatch browsing the children tree from the root UI element: the %@ is
            a child of %@ <%p>, but it has a different parent, %@ <%p>.",
            [element AXRoleDescription], [browserParent AXRoleDescription],
            browserParent, [[element AXParent] AXRoleDescription],
            [element AXParent]);
    } else {
        [cell setValue:[self descriptionWithTitleAndIndexofElement:element
            atColumn:column]];
    }
    [cell setLeaf:[element childrenCount] == 0];
}
}

```

The PFUIElement and PFApplicationUIElement classes declare several other methods relating to attributes. Read the *PFAssistive Framework Reference* for comprehensive information about them.

Controlling an Application

There are three ways in which an active assistive application can control a target application using the Accessibility API:

1. An assistive application can determine whether a UI element of interest has an Accessibility attribute that can be set. If it does, the assistive application sets the attribute to a new value, and the effect takes place immediately in the target application and on the screen. Read the [Setting a UI Element's Attributes](#) subsection for details.
2. An assistive application can determine whether a UI element of interest responds to any of several Accessibility actions. If it does, the assistive application directs the target application to perform the action, and the effect takes place immediately in the target application and on the screen. Read the [Performing Actions on a UI Element](#) subsection for details.
3. An assistive application can send a keystroke, optionally combined with modifier keys, to the target application. If the target application recognizes the key combination as a keyboard shortcut, it performs the shortcut.

immediately. Otherwise, if the UI element in the target application that currently has keyboard focus can respond to the keystroke, it responds immediately, for example, by typing a character into the focused text field or text area. Read the [Sending Keystrokes to an Application](#) subsection for details.

Whenever an assistive application controls a target application using any of these techniques, the target application responds exactly as it would if a user had done the same thing using the graphical user interface. The target application's internal data store and its user interface are updated appropriately, its documents are marked dirty as appropriate, and its undo mechanism takes note so that the user can undo whatever changes were made.

There is a fourth mechanism by which an assistive application can control a target application, namely, by manipulating or creating user input events using the PFiddlesoft PFEVTaps Framework. The PFEVTaps Framework is discussed in the next chapter, [How to Use the PFEVTaps Framework](#).

1. Setting a UI Element's Attributes

The values of many of a UI element's attributes can be set by an assistive application. In the parlance of Objective-C, their properties in the Attributes category of the PFUIElement class are declared **readwrite**, not **readonly**. Setting a settable attribute is a common means used by active assistive applications to control a target application.

Before attempting to set an attribute, an assistive application should test whether it is marked as settable in the Accessibility API. To do this, send it an `-isSettableAttribute:` message, passing in the name of the attribute of interest. Beware that some attributes are marked as settable but their values cannot in fact be set. Code defensively to guard against this possibility.

This statement is used in UI Browser as the first step in marking an attribute as settable in the Attributes drawer:

```
[tempDictionary setObject:[NSNumber numberWithInt:
    [[self currentElement] isSettableAttribute:attribute]] forKey:@"settable"];
```

To set the value of a settable attribute by name, use a standard setter method. Some attributes are always settable and need not be tested first, like this:

```
[myWindowElement setAXMinimized:[NSNumber numberWithInt:YES]];
```

Just as an assistive application can read an unknown attribute using Key-Value Coding (KVC), it can also set the value of an unknown attribute using KVC. As when reading unknown attributes, this works only with non-parameterized attributes. To do this, call PFUIElement's `-setValue:forAttribute:` method. The example on the next page is part of UI Browser's code to set the target application's attributes when the user edits the existing values in the Attributes drawer.

```

- (void)commitEdits {
    PFBrowserController *controller = [self browserController];
    NSTableView *table = [controller attributeTable];
    if ([[self currentElement] isSettableAttribute:[self selectedAttribute]]) {
        if ([[self currentElement] typeForAttribute:[self selectedAttribute]]
            isEqualToString:@"string"]) {

            // Set attribute table to new string.
            [[self currentElement] setValue:[self bottomTextView] string]
                forAttribute:[self selectedAttribute]];
            (NSMutableDictionary *)[controller cachedAttributeArray]
                objectAtIndex:[table selectedRow] setObject:
                [[self bottomTextView] string] forKey:@"value"];
            (NSMutableDictionary *)[controller cachedAttributeArray]
                objectAtIndex:[table selectedRow] setObject:
                [controller descriptionOfAttributeValue:[self bottomTextView] string]
                ofType:@"string" element:[self currentElement]]
                forKey:@"valueDescription"];
            [table reloadData];

            // Set string textfield to new string.
            NSString *selectedAttributeValue =
                [[self currentElement] valueForKey:[self selectedAttribute]];
            [[controller settingStringTextField] setStringValue:selectedAttributeValue];
            [[controller settingStringTextField] selectText:table];
        }
    }
}

```

The `-setValue:forAttribute:` method returns **YES** if the attribute's value was successfully set, or **NO** if not or if the given attribute is a parameterized attribute. However, some applications mark attributes as settable when they are not, and this method may return **YES** even though the value was not changed. Code defensively to guard against this possibility.

There are some cases where you might think that a UI element's value attribute should be settable, but it isn't. In those cases, the UI element can be modified by performing an Accessibility action on it, instead. A common example that often trips up users of GUI Scripting is a checkbox. Apple advises that a checkbox should be modified only by sending it an "AXPress" action simulating a click in the graphical user interface, not by setting its **AXValue** attribute. An assistive application can nevertheless read its **AXValue** attribute. In many cases, however, the target application can be controlled either by setting an attribute or by sending an action. For example, a window can be minimized to the Dock by setting the window element's **AXMinimized** property, as shown previously, or by sending an "AXPress" action to its minimize button element, as shown in the next subsection.

2. Performing Actions on a UI Element

Many UI elements in a target application can be controlled by instructing them to perform one of a handful of Accessibility actions. The Accessibility API recognizes these seven actions, each of which is a string:

- “AXCancel” - a cancel action, such as pressing the Cancel button.
- “AXConfirm” - a confirm action, such as pressing Return in a text field.
- “AXDecrement” - a decrement action, such as pressing a stepper's down arrow.
- “AXIncrement” - an increment action, such as pressing a stepper's up arrow.
- “AXPress” - a press action, such as clicking a button or a menu.
- “AXRaise” - a raise action, such as bringing a window to the front within the application.
- “AXShowMenu” - a show menu user action, such as opening a pop-up button's menu or a contextual menu in a text view or text field.

Applications may respond to custom actions, as well, such as the “AXOpen” action in the Finder.

To get a list of the actions that a particular UI element recognizes, send it the `-actions` message. It returns an array of `NSStrings`. To determine whether an element responds to a given action, send it the `-existsAction:` method, passing in the name of the action. To cause the target application to perform the action on the element, send the element the `-performAction:` method, passing in the name of the action.

This example is the UI Browser action method that is sent to the target application when the user double-clicks any UI element in UI Browser's main browser view. The action method brings the target application and the window containing the UI element to the front and highlights the UI element with a yellow overlay. It starts by activating the target application using a `PFUIElement` utility method, `-activateApplication`. Then, if the UI element that was double-clicked is a window element, it performs the “AXRaise” action on it to make it the target application's frontmost window. If it is any other element, it gets the element's `AXWindow` property and then performs the “AXRaise” action on that.

```
- (IBAction)highlightDoubleClickAction:(id)sender {
    if ([self isTargetAccessible]) {
        [[self currentElement] activateApplication];
        if ([[self currentElement] isRole:NSAccessibilityWindowRole]) {
            [[self currentElement] performAction:NSAccessibilityRaiseAction];
        } else {
            [[[self currentElement] AXWindow]
             performAction:NSAccessibilityRaiseAction];
        }
        [[self highlightButton] setState:NSOnState];
        [self highlightAction:sender];
    }
}
```

3. Sending Keystrokes to an Application

The final way to control a target application using the PFAssistive Framework is to send the application a keystroke, optionally combined with modifier keys. To send a keystroke to a specified application, use `PFAApplicationUIElement's -typeCharacters:keyCode:withModifierFlags:` method. It posts keyboard events to the application represented by the receiving `PFAApplicationUIElement` even if it is not currently the active application. To send a keystroke to the active application without first determining which application is active, send the `+typeCharactersSystemWide:keyCode:withModifierFlags:` class method. It creates a temporary system-wide element and posts keyboard events to the frontmost application.

Both methods are sent directly to the application, not to individual UI elements. They differ from the other techniques for controlling a target application in this respect. Usually, the target application types the character into whatever UI element currently has keyboard focus, just as typing on the keyboard would do. Therefore, an assistive application should set focus to a specific text field or text area first, using the `AXFocused` property, by sending a `-setAXFocused:` message to the element.

Both methods send a single keystroke with optional modifier keys. Apple's corresponding GUI Scripting commands for AppleScript—the 'keystroke' command and the 'keycode' command—are different in this regard; they send multi-character strings by calling the Accessibility API function underlying these PFAssistive Framework methods multiple times. An assistive application can accomplish the same result that GUI Scripting does by similarly sending either of these methods multiple times in succession. However, it is often easier and more efficient to set the `AXValue` attribute of a UI element if it is a text field or a text area. An assistive application can set the value of a text field or a text area by passing an `NSString` of arbitrary length in a `-setAXValue:` or `-setValue:forAttribute:` method.

If the `-typeCharacters:keyCode:withModifierFlags:` or `+typeCharactersSystemWide:keyCode:withModifierFlags:` method specifies that the Command key is down, and if the target application recognizes the key combination as a keyboard shortcut, it executes the shortcut. It is important to pass the command character as a lowercase letter, because an uppercase character is interpreted as a keyboard shortcut with the Shift key down. In some applications this performs a different keyboard shortcut, but in most applications it either does nothing or it types the uppercase character into the active UI element.

The `characters` and `flags` arguments are equivalent to those used in `NSEvent's -charactersIgnoringModifiers` and `-modifierFlags` methods. The `virtualKey` argument is the virtual key code provided by `NSEvent's -keyCode` method. It is a hardware-independent integer value provided by system resources for every known keyboard, mapped from the hardware-dependent raw key code using the current keyboard layout resource.

On Roman systems, the `characters` argument is optional and should be passed as `nil` or an empty string unless you are knowledgeable regarding the difficult and arcane subject of keyboard layouts. It is not optional on some other systems where it is used as a hint to supplement the virtual key code during key translation.

Miscellaneous

There are a few methods in the PFAssistive Framework that have not been covered in this *Programming Guide*. They are covered in detail in the *PFAssistive Framework Reference*.

How to Use the PFEventTaps Framework

Apple's Quartz Event Taps API implements the three concepts of an *event tap* to monitor and intercept user input events; an *event source*, such as a mouse, keyboard, scroll wheel, tablet, or tablet pointer, or a virtual input device; and a user input *event* that is generated by an event source.

The PFEventTaps Framework implements these same concepts in its PFEventTap, PFEventSource, and PFEvent classes, each of which instantiates and encapsulates an associated Event Taps API object and makes its capabilities available to an assistive application using standard Objective-C and Cocoa techniques. For example, an assistive application using the PFEventTaps Framework can implement optional delegate methods declared in the framework's PFEventTapsDelegate formal protocol to observe Quartz events as they are generated by user input devices and virtual devices, and to filter, modify, block, and respond to the events.

In this chapter, you learn how to monitor user input events generated by devices such as a mouse, a keyboard, or a tablet, and to filter, modify, block, and respond to user input events and to generate additional synthetic events, all in the interest of supporting assistive devices and applications that enable a user with disabilities to use the computer to perform the same tasks that any user can perform. An assistive application typically does this by performing these tasks:

- Creating and installing a PFEventTap object that intercepts user input events at one of several points in the system's low-level event handling machinery. It sends delegate messages or callbacks to an assistive application to enable it to filter, modify, block, and respond to the events. Read the [Monitoring Events Using an Event Tap](#) section for more information.
- Retrieving a PFEvent object that represents a user input event generated by an event source and reported to an assistive application by an installed event tap. An assistive application uses the PFEvent object to filter, modify, block, and respond to the event. It can respond to the event by sending additional synthetic events before or after the received event and by taking other actions. Read the [Filtering, Modifying, Blocking, and Responding to Events](#) section for more information.
- Creating PFEvent objects that post independent synthetic user input events, for example, from a virtual onscreen keyboard. Read the [Posting Synthetic Events](#) section for more information.
- Retrieving or creating a PFEventSource object that represents a user input device such as a mouse, keyboard, scroll wheel, tablet, or tablet pointer, or a virtual input device. An event source reports current state information about the associated device outside of the event stream. Read the [Reading the State of an Event Source](#) section for more information.

IMPORTANT WARNING: When an assistive application initializes a PFEventTap object as an active filter and implements an `-eventTap:willPostModifiedEventForEvent:` delegate method or a callback method, no user input event will be delivered to any running application or to the target application unless the method returns a valid PFEvent object. Implementing an `-eventTap:shouldPostEvent:` method and returning `NO` will also block events. It is therefore

prudent during development to initialize an event tap as a passive listener until you are confident that it is returning a valid `PFEEvent` object or that it is blocking only the intended types of events. If the mouse or keyboard becomes unresponsive because the delegate or callback method blocks user input events, use the Option-Command-Escape key combination to force the application to quit or, if keyboard events are blocked, turn off the computer and reboot.

Monitoring Events Using an Event Tap

An assistive application using the `PFEEventTaps` Framework monitors Quartz events using `PFEEventTap` objects. Creating and installing a `PFEEventTap` installs one or more Quartz event taps that monitor hardware user input events from an input device such as a keyboard, mouse, scroll wheel, tablet, or tablet pointer, or synthetic user input events generated in software. A Quartz event tap monitors events at one of several levels in the system, enabling an assistive application to observe and manipulate events that are targeted at any running application or at a specific application in real time as they occur.

A `PFEEventTap` may be configured to respond to one or more types of user input events, such as key up and left mouse down events. When it is triggered, it calls a delegate method or a callback method implemented by the assistive application, passing the `PFEEventTap` itself and a `PFEEvent` object representing the triggering event. It calls some of the delegate methods before the system delivers the event to its intended target, and some after. The assistive application can do anything in response to the event, before or after the target application responds to it. If the `PFEEventTap` is configured as an active filter rather than a passive listener, the assistive application can deliver a modified version of the original event to the original target or to another target. It can also post additional synthetic events, and it can block the original event altogether. If the event tap is configured as a passive listener, the assistive application cannot prevent the original event from being delivered to the intended target, but it can post additional compatible events before and after the original event.

The system guarantees that events which are not blocked are delivered sequentially. However, if an assistive application installs an active `PFEEventTap` and the delegate method or callback method takes too long to execute, the system might become bogged down. For this reason, the system may automatically disable any event tap if it detects excessive delay. An assistive application can detect when the system disables an event tap and re-enable it in order to continue monitoring events.

To monitor Quartz events in your assistive application, create a `PFEEventTap` object using one of the factory convenience methods declared in `PFEEventTap.h`, or create a `PFEEventTap` object and initialize it with one of the declared initializers. It begins monitoring events immediately. Alternatively, create an empty event tap by initializing it with `-init`, then install a Quartz event tap in it later with one of the provided installation methods.

A `PFEEventTap` has a choice of two techniques to watch for and respond to generated events: a delegate method and a callback method. The delegate method is easier to use, and it suffices for most purposes. The callback method allows for more complex behavior when circumstances require it. The choice is made when the `PFEEventTap` is created or a Quartz event tap is installed in an empty event tap.

To create a `PFEEventTap` that uses a delegate method to monitor a specific application, call one of the factory methods for delegates, passing in an event types mask specifying the kinds of events to monitor and the name (not the path) of the target application. Also take one or two additional steps: set the delegate, and in the case of an empty event tap install a Quartz event tap by calling one of the provided installation methods. The delegate must declare that it conforms to the `PFEEventTapsDelegate` formal protocol, which is declared in `PFEEventTaps.h`. Set the delegate by sending a `-setDelegate:` message. Two factory methods for delegates are provided, `+activeTapWithEventTypesMask:`

forApplication: and **+passiveTapWithEventTypesMask:forApplication:**. An *active* event tap can read, modify, and block events, while a *passive* tap can only read events.

This example is the method used in UI Browser to create the `PFEEventTap` object that monitors movement of a target application's window or other UI element so that the highlight overlay can move with it. It uses a utility class method of the `PFEEventTap` class, **+eventTypesMaskByAddingType:toMask:** to create the mask argument to the factory method.

```
- (void)makeHighlightedElementEventTap {
    // Called by -highlightAction: to create a Core Graphics event tap to monitor left
    mouse drags, mousedown and mouseups. Dragging the mouse or clicking an up or down
    button in a scroll bar, if it results in moving or resizing the highlightedElement,
    will also move or resize the overlay highlighting window synchronously by invoking the
    eventTap:didPostEvent: delegate method. Dragging or clicking anything that does not
    move or resize the highlightedElement simply puts the overlay window where it already
    is, to no visual effect.

    unsigned mask =
        [PFEEventTap eventTypesMaskByAddingType:NSLeftMouseDown toMask:0];
    mask = [PFEEventTap eventTypesMaskByAddingType:NSLeftMouseDown toMask:mask];
    mask = [PFEEventTap eventTypesMaskByAddingType:NSLeftMouseUp toMask:mask];
    NSString *appName =
        [[NSFileManager defaultManager] displayNameAtPath:[self currentPath]];
    [self setHighlightedElementEventTap:
        [PFEEventTap passiveTapWithEventTypesMask:mask forApplication:appName]];
    [[self highlightedElementEventTap] setDelegate:self];
}
```

Instead of monitoring a specific target application, an assistive application can monitor all running applications by calling one of the factory methods for delegates that takes a system location instead of an application name, **+activeTapWithEventTypesMask:forLocation:** and **+passiveTapWithEventTypesMask:forLocation:**. The system locations supported by the `PFEEventTaps` Framework are `kCGSessionEventTap` (1), where HID system and remote control events enter a login session, and `kCGAnnotatedSessionEventTap` (2), where session events have been annotated to flow to an application. The `PFEEventTaps` Framework does not support `kCGHIDEEventTap` (0), where HID system events enter the window server and the application must run as root.

To create a `PFEEventTap` that uses a callback method instead of a delegate method, call one of the factory methods for callbacks, passing in an event types mask specifying the kinds of events to monitor and the name (not the path) of the target application, along with an object to serve as the notification delegate and the selector for the callback method that the notification delegate implements. The notification delegate and callback selector follow the rules described in [Observing Notifications](#) in the [How to Use the PFAssistive Framework](#) chapter. Information can also be passed in the `contextInfo` argument that is needed when the callback is called.

None of the factory methods in the `PFEEventTaps` Framework use the target application's BSD Unix process identification number (PID) to identify the target application, although the designated initializers do.

In addition to the factory methods, there are initializers for use when factory methods are inappropriate or the application needs finer control over the event tap. Two of the initializers take the target application's full path (not its name), two take its PID, and two take a system location. Three are for delegate-based event taps, and three for callback-based event taps. The designated initializer for a PFEEventTap object for a specified application is **-initWithEventTypesMask:forApplicationPid:appendAtTail:listenOnly:notificationDelegate:callbackSelector:contextInfo:**. The designated initializer for a PFEEventTap object for a specified system location is **-initWithEventTypesMask:forLocation:appendAtTail:listenOnly:notificationDelegate:callbackSelector:contextInfo:**. The initializers allow an application to specify whether an event tap is appended at the tail or inserted at the head, while the factory methods always use the default, insert at head. The initializers return `nil` if initialization fails or if the event tap is not permitted to monitor the events specified by the event types mask (key up and key down events require that access for assistive events be enabled).

All of the factory methods and initializers configure and install a fully-functional event tap that begins monitoring events immediately. If an assistive application needs to stop monitoring events for any reason, it has two options. It can disable the tap temporarily and reenable it using its **enabled** property by sending a **-setEnabled:** message and passing **YES** or **NO**. Alternatively, it can uninstall the event tap by calling the **-uninstall** method. Uninstalling an event tap leaves the PFEEventTap object in existence as an empty object; release it to reclaim its memory. An assistive application can create an empty PFEEventTap object in the first place by creating it and using **-init** to initialize it. Turn an empty event tap back into a fully configured tap that monitors events by calling one of the three provided installation methods with appropriate configuration arguments.

The example on the next page is the method used in the Event Taps Testbench utility to install a Quartz event tap in an empty PFEEventTap object. Event Taps Testbench subclasses PFEEventTap so that it can associate a name and description with every event tap. It creates an empty PFNamedEventTap object with `[[PFNamedEventTap alloc] init]`. It then obtains configuration values from the user in a window and installs the event tap by calling this method. The instance variables that hold the configuration values are bound to the window's controls using Cocoa bindings.

```

- (void)doInstall:sender {
    // Set up initialization arguments.
    NSUInteger mask = [[self eventTap] tapEventMask];
    NSUInteger location = [[self eventTap] tapLocation];
    NSString *path = [[NSWorkspace sharedWorkspace] fullPathForApplication:[self
eventTap] tapApplicationName]];
    BOOL appendAtTail = ([[self eventTap] tapPlacement] == kCGTailAppendEventTap);
    BOOL listenOnly = ([[self eventTap] tapOption] == kCGEventTapOptionListenOnly);
    id notificationDelegate =
        ((([self eventTap] tapUsesCallback)) ? [self mainWindowController] : nil);
    SEL callbackSelector = ((([self eventTap] tapUsesCallback)) ?
        @selector(eventTap:eventTapProxy:eventType:event:contextInfo:) : NULL);
    NSString *info = ((([self eventTap] tapUsesCallback] &&
        [[self eventTap] tapUsesContextInfo]) ?
        [[self eventTap] tapContextInfo] : @"");

    // Remove old event tap, if any.
    if ([self replacing]) {
        [[self eventTap] uninstall];
        [[self eventTap] setDelegate:nil];
        [[[self mainWindowController] installedTapsController]
            removeObject:[self eventTap]];
        [self setEventTap:[self eventTap]];
    }

    // Install new event tap.
    PFNamedEventTap *newTap = [self eventTap];
    if (location == 3) { // application
        [newTap setIsInstalled:[newTap installWithEventTypesMask:mask
            forApplicationPath:path appendAtTail:appendAtTail listenOnly:listenOnly
            notificationDelegate:notificationDelegate
            callbackSelector:callbackSelector contextInfo:info]];
    } else {
        [newTap setIsInstalled:[newTap installWithEventTypesMask:mask
            forLocation:location appendAtTail:appendAtTail listenOnly:listenOnly
            notificationDelegate:notificationDelegate
            callbackSelector:callbackSelector contextInfo:info]];
    }

    if ([newTap isInstalled]) {
        // Set the new event tap's delegate.
        [newTap setDelegate:([newTap tapUsesDelegate]) ?
            [self mainWindowController] : nil];
        // ....
    }
}

```

Read the [Filtering, Modifying, Blocking, and Responding to Events](#) section for information about what an assistive application can do in its implementation of a delegate or callback method.

Filtering, Modifying, Blocking, and Responding to Events

To respond to an event that was intercepted by an event tap, an assistive application usually uses a delegate and implements one or more of the optional delegate methods declared in the `PFEventTapsDelegate` formal protocol declared in `PFEventTap.h`. Alternatively, an assistive application can implement an Objective-C callback method.

One of the delegate methods, `-eventTap:wasDisabledBy:`, monitors whether the system or the user has disabled the event tap. This delegate method returns the event tap that was disabled, and it reports in the `source` argument whether it was disabled by the system, `kCGEventTapDisabledByTimeout` (0xFFFFFFFF) or by the user, `kCGEventTapDisabledByUserInput` (0xFFFFFFFF). An appropriate response when the system disables it is usually to re-enable it immediately by sending a `-setEnabled:` message, passing `YES`. An assistive application that wants to ensure that it does not fall out of touch with events should always implement this delegate method. These are referred to as *out-of-band events*; they are never posted to the target application.

This example from Event Taps Testbench simply reports to the user when the system disables an event tap:

```
- (void)eventTap:(PFEventTap *)tap wasDisabledBy:(NSUInteger)source {
    if (source == kCGEventTapDisabledByTimeout) { // automatically disabled by timeout
        [self alertForTimeoutDisabledEventTap:(PFNamedEventTap *)tap];
    }
}
```

Three of the other delegate methods follow the standard Cocoa pattern: `-eventTap:willPostEvent:` is called when an event tap detects a user input event but before the event is posted to the target application; `-eventTap:shouldPostEvent:` is called when an event tap detects a user input event but before the event is posted to the target application; and `-eventTap:didPostEvent:` is called when an event tap detects a user input event and after the event is posted to the target application.

Two delegate methods can be used to post a modified event or to block an event if the event tap is configured as an active tap. If the `-eventTap:shouldPostEvent:` delegate method returns `NO`, it blocks the event so that it is never posted to the target application. A special delegate method, `-eventTap:willPostModifiedEventForEvent:`, is called when an event tap detects a user input event but before the event is posted to the target application. The unmodified event is passed in the `event` argument, and the method posts and returns a modified version of the event. If the modified event is `nil`, the effect is the same as returning `NO` from the `-eventTap:shouldPostEvent:` method.

If more than one of the delegate methods is implemented, `PFEventTaps` follows a simple precedence rule. Delegate methods, if implemented, have the precedence from highest to lowest given here: `-eventTap:shouldPostEvent:`, `-eventTap:willPostEvent:`, `-eventTap:willPostModifiedEventForEvent:`, and `-eventTap:didPostEvent:`. Implementing one of them suppresses all with lower precedence, except that the `-eventTap:didPostEvent:` method is always called if an event was posted.

This example is a portion of Event Taps Testbench's implementation of the `PFEventTap`'s `-eventTap:willPostModifiedEventForEvent:` delegate method:

```

- (PFEvent *)eventTap:(PFEventTap *)tap
    willPostModifiedEventForEvent:(PFEvent *)event {
    // ....
    if (doBlockEvent) return nil;
    if (doModifyEvent) {
        PFEvent *modifiedEvent =
            [ (PFNamedEventTap *)tap modifiedEventForEvent:event withProxy:NULL];
        [client updateMonitorEventsWindowWithEvent:modifiedEvent andContextInfo:nil];
        [client updateEventsWindowWithTriggerMethod:@"WILL POST MODIFIED EVENT FOR
            EVENT delegate method" event:modifiedEvent andContextInfo:nil];
        return modifiedEvent;
    }
    // ....
}

```

As an alternative to these delegate methods, an assistive application can implement an Objective-C callback method using one of the factory methods, initializers or installation methods that include parameters for a callback (notification delegate, callback selector, and context info). Use a callback method for more complex scenarios, for example, where the assistive application must use the information saved in the `contextInfo` argument when the event tap was installed. The delegate methods do not have enough information to do this. The assistive application's Objective-C callback selector must have the following signature:

```

- (PFEvent *)eventTap:(PFEventTap *)tap eventTapProxy:(CGEventTapProxy)proxy
    eventType:(unsigned)type event:(PFEvent *)event contextInfo:(void *)info

```

One situation in which an assistive application must use a callback method instead of a delegate method is where it posts additional events that are compatible with the original event using `-postEvent:withProxy:`. The `proxy` argument to be used in this method is an opaque object passed to the callback method to identify the event tap that generated the event, encoded as an NSData object. It provides information to ensure that the posted event is compatible with the event to be returned by the callback method.

This method from Event Taps Testbench is called from a callback method, where it obtained the `proxy` argument, every time the user types a letter on the keyboard. It calls `-postEvent:withProxy:` to type the same letter again, doubling every character typed into any application that is running. Note the steps taken to avoid unwanted side effects, such as disabling the event tap temporarily so that sending the event again does not invoke the callback again in an infinite regression.

```

- (PFEvent *)modifiedEventKeyboardDoubleCharactersForEvent:(PFEvent *)event
    proxy:(CGEventTapProxy)proxy {
    if (![event isCommandKeyDown]) { // don't double keyboard shortcuts
        [self setEnabled:NO]; // avoid infinite regression
        [PFNamedEventTap postEvent:event withProxy:proxy]; // post the same event again
        [self setEnabled:YES];
    }
    return event;
}

```

For good measure, a PFEventTap object posts two notifications that an assistive application can register to observe. The `PFEventTapWillFireNotification` is posted before any delegate or callback method is called, and the `PFEventTapDidFireNotification` is posted after they are called (even after the `-eventTap:didPostEvent:` delegate method, if it is implemented). In both cases, the `notification` object is the PFEventTap object and the `userInfo` dictionary contains a reference to a PFEvent object keyed to "PFEvent". In the first notification, the event is the original event. In the second notification, it is the event that was actually posted. The event that was actually posted may be the original event, if the original event was posted; a modified event, if a modified event was posted; an event tap disabled event, in which case no event was posted; or if `-eventTap:shouldPostEvent:` returned `NO` or `-eventTap:willPostModifiedEventForEvent:` or the callback method returned `nil`, in which case no event was posted. These notifications may be useful if the application needs to do any preparatory or cleanup work before or after the delegate methods are called.

The principal delegate methods and the callback method include an `event` argument that is a PFEvent object. In addition, the PFEvent class implements several factory methods and initializers that an assistive application can use to create new PFEvent objects of any kind, as described in the [Posting Synthetic Events](#) section. The PFEvent class declares a large number of properties that allow an assistive application to read or modify the features of any detected input event, and to set the features of a new synthetic event before posting it.

For example, events of any kind can get a `timestamp` reporting the time when the event was generated in nanoseconds since system startup. Keyboard, mouse, and tablet pointer events can get the `position` of the mouse or pointer on the screen as well as the `modifierKeyFlags` to determine the user's intent while clicking or typing. Mouse events can get the `mouseClickState` to distinguish single-click and double-click events, the `mouseButtonNumber` to learn whether it was a left mouse or right mouse event or a fire button event on a joystick, `mouseDeltaX` and `mouseDeltaY` to know how far the mouse has moved since the last mouse event, and many others. A keyboard event reports whether it `isKeyAutorepeat` and others. Scroll wheel events are covered, as are tablet proximity and tablet pointer events both for mouse events with a tablet subtype and for pure tablet events. Consult the *PFEventTaps Framework Reference* for complete information.³ All of these PFEvent properties are available to an assistive application to help it respond to user input events in any application.

Posting Synthetic Events

In addition to reacting to user input events, an assistive application can generate synthetic user input events and post them independently. In reality, all synthetic events are posted in response to a user input event, because it takes some signal from the user to tell the assistive application to post another event. The important distinction is between events

³ This version of the PFEventTaps Framework does not cover multi-touch or gesture events.

that modify or supplement a user input event, which are typically posted using the `-postEvent:withProxy:` method discussed above, and events that in some way emulate a hardware input device in software. For example, a virtual keyboard on the screen might respond to a user's mouse clicks by generating key down and key up synthetic keyboard events, and the iSight camera on the monitor might interpret the user's gestures in space and generate synthetic mouse moved events.

To post a synthetic event from your assistive application, first create a `PFEvent` object using one of the factory convenience methods declared in `PFEvent.h`, or create a `PFEvent` object and initialize it with one of the declared initializers. To create any kind of event, create it and initialize it with the `-initWithEventSource:` initializer or create and return it with the `+eventWithEventSource:` factory method. To create a keyboard, mouse, or scroll wheel event, use one of the specific factory methods or their corresponding initializers designed for this purpose, passing in state information such as the `keyCode` and `keyDown` arguments to the `+keyboardEventWithEventSource:keyCode:keyDown:` factory method, the `type`, `position`, and `button` arguments to the `+mouseEventWithEventSource:type:position: button:` factory method, and the `scrollUnits`, `count`, and `wheelRanges` arguments to the `+scrollWheelEventWithEventSource: scrollUnits:wheelCount:wheelRanges:` factory method.

All of these methods require that a `PFEventSource` object be passed in as the `source` argument. If an assistive application does not use an event source that is already available, it must create the event source itself. An existing `PFEventSource` object can be obtained from the `PFEvent` object passed into an event tap's delegate method or callback method by getting the `PFEvent`'s `eventSource` property. To create a new event source, an assistive application must use one of the `PFEventSource` factory convenience methods or initializers declared in `PFEventSource.h`. See the [Reading the State of an Event Source](#) section for more information.

There are several other factory methods and initializers that do not require an event source to create an event. The `+eventWithEventRef:` and `+eventWithCopyOfEventRef:` factory methods and their corresponding initializers take a Core Graphics `CGEventRef` object as their event arguments. The `+eventWithEventData:` factory method and its corresponding initializer take an `NSData` object that is a flattened data representation of an event received over a network or created from a `PFEvent` object for transmission over a network using `PFEvent`'s `-eventData` method.

The designated initializer for `PFEvent` is `-initWithEventRef:`.

Once an assistive application has created a `PFEvent` object and configured it using `PFEvent`'s properties, it can post the event to an application or to a system location using `PFEvent`'s `-postToApplication:` or `-postAtTapLocation:` method. The event is posted immediately, and it passes through all event taps installed for the application or at the location. To post an event at a location, specify `kCGSessionEventTap` (1), where HID system and remote control events enter a login session, or `kCGAnnotatedSessionEventTap` (2), where session events have been annotated to flow to an application. An assistive application can use `-postAtTapLocation:` to establish an event routing policy, for example, by tapping and blocking events at the `kCGAnnotatedSessionEventTap` location and posting them to another application. This might be useful in an assistive application that provides an alternate user interface to help users with disabilities control an existing application.

Although `PFEvent`'s `position` property is writable, special techniques provided by Quartz Display Services are required to move the cursor on the screen so that its visible position corresponds to its modified `position` property. For mouse and tablet pointer events, an assistive application must first dissociate the cursor from system control by calling `CGAssociateMouseAndMouseCursorPosition(false)`. To reposition the cursor, it must then call `CGDisplayMoveCursorToPoint()` or `CGWarpMouseCursorPosition()` to move the cursor. Finally, to restore

normal cursor functioning, an assistive application must call `CGAssociateMouseAndMouseCursorPosition(true)`. See Apple's *Quartz Display Services Programming Topics* for more information. `PFEvent` covers these functions with its `+dissociateCursor`, `+warpCursorToPoint:`, and `+associateCursor` class methods. Use the `mouseDeltaX` and `mouseDeltaY` properties to learn how far the cursor's position has moved since the last mouse down or mouse dragged event was posted; those properties continue to register changes even while the cursor is dissociated.

Keep these relations and requirements in mind when moving the cursor:

- `NSEvent`'s `+mouseLocation` class method reports the location where the cursor is now, outside the event stream, before a mouse dragged or mouse moved event is posted and before the cursor has moved.
- `PFEvent`'s `position` property reports where the cursor will be after the event is posted.
- `PFEvent`'s `mouseDeltaX` and `mouseDeltaY` properties report how far the cursor will have moved after the event is posted.

If `CGAssociateMouseAndMouseCursorPosition(false)` has been called, `PFEvent`'s `position` property remains frozen but its `mouseDeltaX` and `mouseDeltaY` properties continue to function as usual. The cursor can be repositioned where desired by calling the `CGWarpMouseCursorPosition()` function, which posts no events. If the cursor is repositioned, it is necessary to reset `PFEvent`'s `position` property to the new cursor position so that user clicks will take effect at the location indicated by the cursor.

This example method is used in Event Taps Testbench as part of the code that freezes the mouse pointer on the screen:

```
- (PFEvent *)modifiedEventMouseFreezeForEvent:(PFEvent *)event {
    if ([event isControlKeyDown]) {
        [self setFrozenCursorPosition:NSZeroPoint]; // signal need to refresh position
    } else {
        if (NSEqualPoints([self frozenCursorPosition], NSZeroPoint)) {
            [self setFrozenCursorPosition:[event position]];
        }
        // Force cursor to move to the frozen position, effectively freezing cursor.
        if ([PFEvent warpCursorToPoint:[self frozenCursorPosition]])
            [event setPosition:[self frozenCursorPosition]];
    }
    return event;
}
```

Reading the State of an Event Source

A `PFEventSource` object is not only used to create a `PFEvent` object, but it is independently useful to read the current state of any standard user input device outside of the event stream, such as whether one of the Option keys on the keyboard is down, whether the right mouse button is down, or the resolution of a scroll wheel in pixels per inch.

Several factory methods and initializers are declared in `PFEventSource.h` for use in creating `PFEventSource` objects. The most useful for reading the state of an input device are `+eventSourceForUserLoginSession`, when creating an

event source if the assistive application is posting events from within a user login session, and **+eventSourceForPrivateSource**, when creating an event source if the assistive application is posting events from a session where events have been annotated to flow to an application. The designated initializer is **-initWithEventSourceRef:**.

An assistive application can determine the state of an input device for its **PFEventSource** object at any time by used one of several properties. These include **isLeftMouseButtonDown**, **modifierKeyFlags**, **isNumericPadKeyDown**, **isCapsLockKeyDown**, **isCommandKeyDown**, **isSecondaryFunctionKeyDown**, and a number of similar properties.

Properties and methods are also provided to determine the time interval since the last user input of any type or all types and the count of events since the window server started.

Miscellaneous

There are a few methods in the **PFEventTaps Framework** that have not been covered in this *Programming Guide*. They are covered in detail in the *PFEventTaps Framework Reference*.

How to Enable Access for Assistive Devices

Before an assistive application can make use of a `PFUIElement`, `PFAApplicationUIElement`, or `PFObserver` object, the "Enable access for assistive devices" setting in the Universal Access pane of System Preferences must be turned on, or the application must be trusted through use of the accessibility API's `AXMakeProcessTrusted()` function. Similarly, an assistive application can monitor key up and key down events using a `PFEVTaps` object only if access is enabled or the application is trusted. Authentication may be required.

IMPORTANT WARNING: The "Enable access for assistive devices" setting is turned off by default due to security concerns. As this *Programming Guide* demonstrates, the Accessibility and Quartz Event Taps APIs are very powerful, allowing an assistive application to control any other application in virtually every respect. The setting applies globally to all applications on the computer. Although the setting can be left on all the time, you should do so only if you are comfortable with the security of your environment.

The most straightforward way to enable access is to open the Universal Access pane of System Preferences and enable it manually. The "Enable access for assistive devices" checkbox appears at the bottom of the pane no matter which tab is selected. In Mac OS X 10.5 Leopard, you can also select the Enable GUI Scripting checkbox in AppleScript Utility. It controls the same system setting. Authentication is required in either case unless you are an administrative user. (AppleScript Utility is a scriptable faceless background application in Mac OS X 10.6 Snow Leopard.)

Access can also be enabled or disabled using an AppleScript script addressed to System Events, a scriptable faceless background application in `/System/Library/CoreServices`. This AppleScript handler automatically presents a dialog where the user can authenticate, and then it enables or disables access depending on the value of the `switch` argument:

```
on enabledGUIScripting(switch)
    tell application "System Events"
        activate
        set UI elements enabled to switch
        return UI elements enabled
    end tell
end enabledGUIScripting
```

An assistive application can enable or disabled access by sending similar AppleScript commands to the System Events application. It can execute the script itself using methods in Cocoa's `NSAppleScript` class or OSKit's `OSAScript` class, or it can use the Scripting Bridge. The details are beyond the scope of this *Programming Guide*.

Finally, an assistive application can arrange to make itself trusted by using the Accessibility API's `AXMakeProcessTrusted()` function. This is by far the best solution from the end user's point of view, because the global "Enable access for assistive devices" setting can be turned off and left off, enhancing overall security. As a trusted

application process, an assistive application can make full use of the Accessibility and Quartz Event Taps APIs, and it remains trusted forever as long as it remains installed.

Making an assistive application trusted takes considerable effort on the part of the developer. As a result, very few applications have taken this route. UI Browser and Event Taps Testbench are two that do use this technique. In outline, these are the requirements:

- Include a very small SUID helper application in the Contents folder of the assistive application's bundle that runs as root and executes the `AXMakeProcessTrusted()` function on the assistive application's main application process.
- Include another very small helper application in the Contents folder that terminates the assistive application and relaunches it, so that it will begin running as a trusted application.
- Add an appropriate user interface to the assistive application whereby the user can authenticate and run the helper applications.
- The assistive application must contain no embedded frameworks, because Apple considers embedded frameworks in a trusted application to be a security concern. This is why the `PFAssistive` and `PFEvtTaps` Frameworks are designed with the expectation that they will be installed in the standard location for shared frameworks, `/Library/Frameworks`. This has the advantage of allowing multiple assistive applications to use the same shared framework, reducing overall memory requirements.
- The assistive application must be installed using an installer package, to ensure that the frameworks are installed in the correct location.

The `PFUIElement` class includes three class methods for use by an assistive application to test the current state of access or trust, `+isAccessImplemented`, `+isAccessEnabled`, and `+isProcessTrusted`.