HID Class Device Interface Guide

Drivers, Kernel, & Hardware: User-Space Device Access

2009-10-19

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

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to Working With HID Class Device Interfaces

What Are HID Class Device Interfaces?

The device interface mechanism supported by the I/O Kit gives applications the ability to communicate with hardware from outside the kernel. This document describes how to use the device interface provided by the Human Interface Device (HID) family to access HID class devices (such as keyboards, mice, and uninterruptible power supplies) from applications running on Mac OS X.

Who Should Read This Document?

You should read this document if you are an application developer who needs to write custom code to communicate with a HID class device from user space.

Important: This document is not intended to cover kernel-level HID device drivers. While it may provide some benefit in terms of understanding HID at a conceptual level, the details of writing in-kernel HID drivers are beyond the scope of this document. For more information on kernel drivers, see *Getting Started with Hardware and Drivers* and *Kernel Framework Reference*.

Organization of This Document

The document is divided into two main chapters:

- "USB HID Overview" (page 9) provides basic information about HID class devices and the Mac OS X HID Manager.
- "Working With Legacy HID Class Device Interfaces" (page 57) briefly outlines the process of accessing a HID class device and then presents a detailed code sample illustrating this process by acquiring access to a joystick.

Although the sample code in this document has been checked for accuracy, it is not intended to meet the needs of a commercial application. For example, error handling is minimal and simply facilitates debugging of this code—you should develop your own techniques for detecting and handling errors. Therefore Apple does not recommend that you directly incorporate the entire sample program into a commercial application.

Important: The sample code in this document is written to work with Mac OS X version 10.3 and later, and may not work with earlier releases.

See Also

This document assumes you are familiar with the general I/O Kit and device interface information presented in *Accessing Hardware From Applications*. In particular, for definitions of I/O Kit terms used in this document such as matching dictionary, family, and driver, see the overview of I/O Kit terms and concepts in the chapter *Accessing Hardware From Applications*.

A detailed description of the HID class specification is beyond the scope of this document—for more information, including the complete listing of HID usage tables, visit the USB website at http://www.usb.org.

For API documentation, see the `IOHIDLib.h` and `IOHIDKeys.h` entries in *I/O Kit Framework Reference*.

USB HID Overview

The Human Interface Device (HID) class is one of several device classes described by the USB (Universal Serial Bus) architecture. The HID class consists primarily of devices humans use to control a computer system's operations. Examples of such HID class devices include:

- Keyboards and pointing devices such as mice, trackballs, and joysticks
- Front-panel controls such as knobs, switches, sliders, and buttons (for example, controls on non-Apple displays)
- Controls that might be found on games or simulation devices such as data gloves, throttles, and steering wheels

Mac OS X provides the HID Manager (described in "The HID Manager" (page 10)) to support access to any devices that conform to the USB HID specification. While this is most commonly used for communicating with input devices, a number of other devices also use HID descriptors, and can thus be accessed using the same mechanism.

For example, you can use the HID Manager to get information from many UPS (uninterruptible power supply) devices. UPS devices share the same report descriptor structure as other HID class devices and provide information such as voltage, current, and frequency. To control a UPS device, however, you access the device's information using HID Manager functions and use it to drive the Power Manager, a process not described in this document.

HID interfaces are also sometimes used as a mechanism to peek and poke small amounts of data when communicating with certain devices that you might not think of as being human–interface-related. For example, there are various generic interface chips designed to provide low-bandwidth control of and input from non-computing devices, such as motor controllers, thermistors, and so on.

HID Class Terminology and Concepts

Information about a HID class device is contained in its HID report descriptors. The report descriptors are provided by the kernel-resident driver of the device and contain descriptions of each piece of data generated by the device. A key component of these report descriptors is the usage information defined in the USB HID Usage Tables. (You can download these from http://www.usb.org/developers/hidpage.) Usage values describe three basic types of information about the device:

- controls—information about the state of the device such as on/off or enable/disable
- data—all other information that passes between the device and the host
- collections—groups of related controls and data

Taken together, the usage page and usage number define a unique constant that describes a particular type of device or part of that device. For example, on the Generic Desktop usage page (page number `0x01`), usage number `0x05` is a game pad and usage number `0x39` is a hat switch.

Logically distinct components of a HID class device such as an x axis, y axis, dial, or slider, are called elements. Information about the elements of a HID class device are grouped into arrays of nested dictionaries. The top or outer level element usually describes the device itself. For example, the top level element for a game pad would include usage page 0x01 (generic desktop) and usage number 0x05 (game pad) followed by an array of other elements. For a game pad that contains both a pointing device and some number of buttons, this array would contain an element for the pointing device and an element for each button. In turn, the element representing the pointing device would probably contain its own array of elements, each representing an axis.

Each element dictionary contains the element cookie (a 32-bit value used to reference that specific element), the usage page and usage number, the collection type, and perhaps other information such as the element's minimum and maximum (for example, an x-axis might have a minimum of -127 and a maximum of 127), and whether or not the element has a preferred state. The element information for all HID class devices currently attached to the running system is available in the I/O Registry so you can check to see if a device has the elements you need before you create a device interface to communicate with it.

The HID Manager

The Mac OS X HID Manager consists of three layers:

- the HID Manager client API that provides definitions and functions your application can use to work with HID class devices
- the HID family that provides the in-kernel HID infrastructure such as the base classes, the kernel-user space memory mapping and queueing code, and the HID parser
- the HID drivers provided by Apple

As an application developer, you will be directly concerned only with the first layer, the HID Manager client API, which this document simply calls the HID Manager. You can access information about the HID Manager from the Sample Code > Hardware & Drivers > Human Interface Device & Force Feedback section of the developer documentation website.

Device Interface Functions

The HID Manager includes `IOHIDManager.h`, `IOHIDLib.h`, and `IOHIDKeys.h` (located in `/System/Library/Frameworks/IOKit.framework/Headers/hid`) which define the property keys that describe a device, the element keys that describe a device's elements, and the device interface functions and data structures you use to communicate with a device. After you've created a device interface for a selected HID class device, you can use the device interface functions to open and close the device, get the most recent value of an element, or set an element value. For the complete list of functions, see `IOHIDLib.h`.

Modern HID manager functions pass around raw HID reports (arrays of bytes). The HID manager provides tools for working with these reports and simple abstraction layers that allow you to perform many tasks without needing to understand the structure of these reports.

Many legacy device interface functions such as `getElementValue`, `getNextEvent`, and `setElementDefault` use a structure called the `IOHIDEventStruct` to contain information about events. (An event is the value of a particular element along with the time it occurred.) These legacy functions that need access to the value of an element use the `IOHIDEventStruct` even though some may ignore the time the value occurred.

Queues

Once you've created a device interface, you can use functions provided by the HID Manager to get the most recent value of an element. For many elements this is sufficient. If, however, you need to keep track of all values of an element, rather than just the most recent one, you can use functions provided by the HID Manager to create a queue and add the element to it. Then, all events involving that element will be contained in the queue (up to the queue depth).

For example, during game play, it's not necessary to keep track of every value of a game device's x and y axis, it's sufficient to update the state of the game to the most recent values of these elements. If there's a "fire" button, however, it's important to respond to every button press, not just the most recent one, so you should add the "fire" button to a queue. Then, every time through the game loop, you can read the queue until it's empty and you won't miss any "fire" events.

Accessing a HID Device

Mac OS X version 10.5 ("Leopard") introduces new APIs that abstract the current complexities of utilizing the I/O Kit to communicate with the HID Manager. These APIs allow you to enumerate HID devices and elements, access their properties, register for notification of HID device discovery and removal (hot plugging and unplugging), send and receive device reports, use queues to get notification of HID element value changes, and use transactions to talk to HID devices. Application developers that need to communicate with HID devices should read this chapter first.

This documentation assumes a basic understanding of the material contained in *Accessing Hardware From Applications*. For definitions of I/O Kit terms used in this documentation (such as matching dictionaries) see the overview of I/O Kit terms and concepts in the Device Access and the I/O Kit chapter. A detailed description of the HID class specification is beyond the scope of this document. For more information, including the complete listing of HID usage tables, visit the USB website at http://www.usb.org/developers/hidpage/.

Note: For binary compatibility, HID code written for Mac OS X version 10.4 ("Tiger") will continue to work for the lifetime of Mac OS X version 10.5 ("Leopard"). New development targeting Leopard should use the new HID Manager APIs.

Note: All HID functions with callback parameters also have a context pointer parameter whose value is passed to that callback. These context pointers are intended for developer use and are passed as-is to the callback routines. In particular, they are not retained, released, or freed in any way. If you need to retain an object passed to one of these routines, you must do so yourself prior to registering the callback.

Device Matching and Access

HID Manager references are used to communicate with the I/O Kit HID subsystem. They are created by using the `IOHIDManagerCreate` function:

```
// Create HID Manager reference
IOHIDManagerRef IOHIDManagerCreate(
                    CFAllocatorRef  inCFAllocatorRef,   // Allocator to be used
 during creation
                    IOOptionBits    inOptions);         // options Reserved for
 future use
```

The first parameter (`allocator`) is a CFAllocator to be used when allocating the returned `IOHIDManagerRef`. The last parameter (`options`) is currently reserved for future use. Developers should pass `kIOHIDOptionsTypeNone` (`zero`) for this parameter.

There is no `IOHIDManagerDestroy` (or release, free, and so on); because the HID Manager reference is a Core Foundation object reference, `CFRelease` should be used to dispose of it.

A `CFTypeRef` can be verified to be a HID Manager reference by comparing its Core Foundation type against `IOHIDManagerGetTypeID`:

Listing 2-1 Validating a HID Manager reference

```
if (CFGetTypeID(tCFTypeRef) == IOHIDManagerGetTypeID()) {
    // this is a HID Manager reference!
}
```

Matching HID Devices

Once a HID Manager reference has been created, it has to be opened before it can be used to access the HID devices associated with it. To restrict the HID devices with which a HID Manager reference is associated, set a matching dictionary (single criteria) or array of matching dictionaries (multiple criteria). The functions are:

```
// Sets single matching criteria (dictionary) for device enumeration.
void IOHIDManagerSetDeviceMatching(
        IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
        CFDictionaryRef inMatchingDictRef); // single dictionary containing
device matching criteria.

// Sets multiple matching criteria (array of dictionaries) for device enumeration.
void IOHIDManagerSetDeviceMatchingMultiple(
        IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
        CFArrayRef      inCFArrayRef);      // array of dictionaries containing
 device matching criteria.
```

Note: Either one of the above APIs must be called before any devices will be matched.

The matching keys used in the dictionary entries are declared in `<IOKit/hid/IOHIDKeys.h>`:

Listing 2-2 HID device property keys

```
#include <IOKit/hid/IOHIDKeys.h>

#define kIOHIDTransportKey                  "Transport"
#define kIOHIDVendorIDKey                   "VendorID"
#define kIOHIDVendorIDSourceKey             "VendorIDSource"
#define kIOHIDProductIDKey                  "ProductID"
#define kIOHIDVersionNumberKey              "VersionNumber"
#define kIOHIDManufacturerKey               "Manufacturer"
#define kIOHIDProductKey                    "Product"
#define kIOHIDSerialNumberKey               "SerialNumber"
#define kIOHIDCountryCodeKey                "CountryCode"
#define kIOHIDLocationIDKey                 "LocationID"
#define kIOHIDDeviceUsageKey                "DeviceUsage"
#define kIOHIDDeviceUsagePageKey            "DeviceUsagePage"
#define kIOHIDDeviceUsagePairsKey           "DeviceUsagePairs"
#define kIOHIDPrimaryUsageKey               "PrimaryUsage"
```

```
#define kIOHIDPrimaryUsagePageKey           "PrimaryUsagePage"
#define kIOHIDMaxInputReportSizeKey         "MaxInputReportSize"
#define kIOHIDMaxOutputReportSizeKey        "MaxOutputReportSize"
#define kIOHIDMaxFeatureReportSizeKey       "MaxFeatureReportSize"
#define kIOHIDReportIntervalKey             "ReportInterval"
```

Note: The `kIOHIDPrimaryUsageKey` and `kIOHIDPrimaryUsagePageKey` keys are no longer rich enough to describe a HID device's capabilities. For example, take a HID device that describes both a keyboard and a mouse in the same descriptor. The previous behavior was to only describe the keyboard behavior with the primary usage and usage page. Needless to say, this would sometimes cause a program interested in mice to skip this device when matching. To resolve this issue three additional keys have been added:

- `kIOHIDDeviceUsageKey`
- `kIOHIDDeviceUsagePageKey`
- `kIOHIDDeviceUsagePairsKey`

The `kIOHIDDeviceUsagePairsKey` key is used to represent an array of dictionaries containing key/value pairs referenced by `kIOHIDDeviceUsageKey` and `kIOHIDDeviceUsagePageKey`. These usage pairs describe all application type collections (behaviors) defined by the HID device.

An application interested in only matching on one criteria would only add the `kIOHIDDeviceUsageKey` and `kIOHIDDeviceUsagePageKey` keys to the matching dictionary. If it is interested in a HID device that has multiple behaviors, the application would instead add an array of dictionaries referenced by `kIOHIDDeviceUsagePairsKey` to their matching dictionary.

This is equivalent to passing an array of dictionaries each containing two entries with keys `kIOHIDDeviceUsagePageKey` and `kIOHIDDeviceUsageKey` to `IOHIDManagerSetDeviceMatchingMultiple`.

Passing a `NULL` dictionary will result in all devices being enumerated. Any subsequent calls will cause the HID Manager to release previously enumerated devices and restart the enumeration process using the revised criteria.

Listing 2-3 Matching against a single set (dictionary) of properties

```
// function to create matching dictionary
static CFMutableDictionaryRef hu_CreateDeviceMatchingDictionary(UInt32
inUsagePage, UInt32 inUsage)
{
    // create a dictionary to add usage page/usages to
    CFMutableDictionaryRef result = CFDictionaryCreateMutable(
        kCFAllocatorDefault, 0, &kCFTypeDictionaryKeyCallBacks,
&kCFTypeDictionaryValueCallBacks);
    if (result) {
        if (inUsagePage) {
            // Add key for device type to refine the matching dictionary.
            CFNumberRef pageCFNumberRef = CFNumberCreate(
                        kCFAllocatorDefault, kCFNumberIntType, &inUsagePage);
            if (pageCFNumberRef) {
                CFDictionarySetValue(result,
                        CFSTR(kIOHIDDeviceUsagePageKey), pageCFNumberRef);
```

```
                CFRelease(pageCFNumberRef);

                // note: the usage is only valid if the usage page is also
defined
                if (inUsage) {
                    CFNumberRef usageCFNumberRef = CFNumberCreate(
                                    kCFAllocatorDefault, kCFNumberIntType,
&inUsage);
                    if (usageCFNumberRef) {
                        CFDictionarySetValue(result,
                            CFSTR(kIOHIDDeviceUsageKey), usageCFNumberRef);
                        CFRelease(usageCFNumberRef);
                    } else {
                        fprintf(stderr, "%s: CFNumberCreate(usage) failed.",
__PRETTY_FUNCTION__);
                    }
                }
            } else {
                fprintf(stderr, "%s: CFNumberCreate(usage page) failed.",
__PRETTY_FUNCTION__);
            }
        }
    } else {
        fprintf(stderr, "%s: CFDictionaryCreateMutable failed.",
__PRETTY_FUNCTION__);
    }
    return result;
}   // hu_CreateDeviceMatchingDictionary

// Create a matching dictionary
CFDictionaryRef matchingCFDictRef =
                    hu_CreateDeviceMatchingDictionary(kHIDPage_GenericDesktop,
 kHIDUsage_GD_Keyboard);
if (matchingCFDictRef) {
    // set the HID device matching dictionary
    IOHIDManagerSetDeviceMatching(managerRef, matchingCFDictRef);
} else {
    fprintf(stderr, "%s: hu_CreateDeviceMatchingDictionary failed.",
__PRETTY_FUNCTION__);
}
```

Listing 2-4 Matching against multiple sets (arrays of dictionaries) of properties

```
// create an array of matching dictionaries
CFArrayRef matchingCFArrayRef = CFArrayCreateMutable(kCFAllocatorDefault, 0,
&kCFTypeArrayCallBacks);
if (matchingCFArrayRef) {
    // create a device matching dictionary for joysticks
    CFDictionaryRef matchingCFDictRef =
                     hu_CreateDeviceMatchingDictionary(kHIDPage_GenericDesktop,
 kHIDUsage_GD_Joystick);
    if (matchingCFDictRef) {
        // add it to the matching array
        CFArrayAppendValue(matchingCFArrayRef, matchingCFDictRef);
        CFRelease(matchingCFDictRef); // and release it
    } else {
```

```
        fprintf(stderr, "%s: hu_CreateDeviceMatchingDictionary(joystick) failed.",
 __PRETTY_FUNCTION__);
    }

    // create a device matching dictionary for game pads
   matchingCFDictRef = hu_CreateDeviceMatchingDictionary(kHIDPage_GenericDesktop,
 kHIDUsage_GD_GamePad);
    if (matchingCFDictRef) {
        // add it to the matching array
        CFArrayAppendValue(matchingCFArrayRef, matchingCFDictRef);
        CFRelease(matchingCFDictRef); // and release it
    } else {
        fprintf(stderr, "%s: hu_CreateDeviceMatchingDictionary(game pad) failed.",
 __PRETTY_FUNCTION__);
    }
} else {
    fprintf(stderr, "%s: CFArrayCreateMutable failed.", __PRETTY_FUNCTION__);
}

-- EITHER --

// create a dictionary for the kIOHIDDeviceUsagePairsKey entry
matchingCFDictRef = CFDictionaryCreateMutable(
    kCFAllocatorDefault, 0, &kCFTypeDictionaryKeyCallBacks,
&kCFTypeDictionaryValueCallBacks);

// add the matching array to it
CFDictionarySetValue(matchingCFDictRef, CFSTR(kIOHIDDeviceUsagePairsKey),
matchingCFArrayRef);
// release the matching array
CFRelease(matchingCFArrayRef);

// set the HID device matching dictionary
IOHIDManagerSetDeviceMatching(managerRef, matchingCFDictRef);

// and then release it
CFRelease(matchingCFDictRef);

-- OR --

// set the HID device matching array
IOHIDManagerSetDeviceMatchingMultiple(managerRef, matchingCFArrayRef);

// and then release it
CFRelease(matchingCFArrayRef);
```

Before opening the HID Manager reference it may be desirable to register routines to be called when (matching) devices are connected or disconnected.

Note: This matching routine is called once per currently connected (and matching) device when the HID Manager reference is opened.

```
// Register device matching callback routine
// This routine will be called when a new (matching) device is connected.
void IOHIDManagerRegisterDeviceMatchingCallback(
```

```
        IOHIDManagerRef     inIOHIDManagerRef,      // HID Manager reference
        IOHIDDeviceCallback inIOHIDDeviceCallback,  // Pointer to the callback
 routine
        void *              inContext);             // Pointer to be passed to
 the callback

// Registers a routine to be called when any currently enumerated device is
removed.
// This routine will be called when a (matching) device is disconnected.
void IOHIDManagerRegisterDeviceRemovalCallback(
        IOHIDManagerRef     inIOHIDManagerRef,      // HID Manager reference
        IOHIDDeviceCallback inIOHIDDeviceCallback,  // Pointer to the callback
 routine
        void *              inContext);             // Pointer to be passed to
 the callback
```

Note: There is no special function to unregister HID callback routines. You can unregistered by calling the appropriate registration function and passing `NULL` for the pointer to the callback routine.

Listing 2-5 Examples of HID device matching & removal callback routines

```
// this will be called when the HID Manager matches a new (hot plugged) HID
device
static void Handle_DeviceMatchingCallback(
            void *          inContext,       // context from
IOHIDManagerRegisterDeviceMatchingCallback
            IOReturn        inResult,        // the result of the matching
operation
            void *          inSender,        // the IOHIDManagerRef for the new
 device
            IOHIDDeviceRef  inIOHIDDeviceRef // the new HID device
) {
    printf("%s(context: %p, result: %p, sender: %p, device: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender, (void*)
inIOHIDDeviceRef);
}   // Handle_DeviceMatchingCallback

// this will be called when a HID device is removed (unplugged)
static void Handle_RemovalCallback(
                void *          inContext,       // context from
IOHIDManagerRegisterDeviceMatchingCallback
                IOReturn        inResult,        // the result of the removing
operation
                void *          inSender,        // the IOHIDManagerRef for the
 device being removed
                IOHIDDeviceRef inIOHIDDeviceRef // the removed HID device
) {
    printf("%s(context: %p, result: %p, sender: %p, device: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender, (void*)
inIOHIDDeviceRef);
}   // Handle_RemovalCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Scheduling the HID Manager on a Run Loop

Before HID Manager callback routines can be dispatched the HID Manager reference must first be scheduled with a run loop:

```
// Schedule HID Manager with run loop
void IOHIDManagerScheduleWithRunLoop(
        IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
       CFRunLoopRef    inRunLoop,         // Run loop to be used when scheduling
 asynchronous activity
        CFStringRef     inRunLoopMode);     // Run loop mode to be used when
scheduling
```

This formally associates the HID Manager with the client's run loop. This schedule will propagate to all HID devices that are currently enumerated and to new HID devices as they are matched by the HID Manager.

Listing 2-6 Scheduling a HID Manager with the current run loop

```
      IOHIDManagerScheduleWithRunLoop(inIOHIDManagerRef, CFRunLoopGetCurrent(),
 kCFRunLoopDefaultMode);
```

There is a corresponding function to unschedule a HID Manager reference from a run loop:

```
void IOHIDManagerUnscheduleFromRunLoop(
        IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
        CFRunLoopRef    inRunLoop,          // Run loop to be used when
unscheduling asynchronous activity
        CFStringRef     inRunLoopMode);     // Run loop mode to be used when
unscheduling
```

Listing 2-7 Unscheduling a HID Manager from a run loop

```
IOHIDManagerUnscheduleFromRunLoop(managerRef, CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode);
```

Now we're ready to open the HID Manager reference.

```
// Open a HID Manager reference
IOReturn IOHIDManagerOpen(
            IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
            IOOptionBits    inOptions);         // Option bits
```

This will open all matching HID devices. It returns `kIOReturnSuccess` if successful. Currently the only valid options (second parameter) are `kIOHIDOptionsTypeNone` or `kIOHIDOptionsTypeSeizeDevice` (which forces exclusive access for all matching devices).

Note: As of Leopard, the `kIOHIDOptionsTypeSeizeDevice` option requires root privileges to be used with keyboard devices.

If there is a device matching callback routine registered when IOHIDManagerOpen is called then this routine will be called once for each HID device currently connected that matches the current matching criteria. This routine will also be called when new devices that match the current matching criteria are connected to the computer (but only if the HID Manager reference is still open).

Listing 2-8 Opening a HID Manager reference

```
// open it
IOReturn tIOReturn = IOHIDManagerOpen(managerRef, kIOHIDOptionsTypeNone);
```

Once a HID Manager reference has been opened it may be closed by using the IOHIDManagerClose function:

```
// Closes the IOHIDManager
IOReturn IOHIDManagerClose(IOHIDManagerRef inIOHIDManagerRef,  // HID Manager
reference
                           IOOptionBits    inOptions);        // Option bits
```

This will also close all devices that are currently enumerated. The options are propagated to the HID device close function.

Registering Value Callbacks

Once a connection to the HID manager is open, developers may register a routine to be called when input values change:

```
// Register a routine to be called when an input value changes
void IOHIDManagerRegisterInputValueCallback(
        IOHIDManagerRef     inIOHIDManagerRef,  // HID Manager reference
        IOHIDValueCallback  inCallback,         // Pointer to the callback
routine
        void *              inContext);         // Pointer to be passed to the
 callback
```

The registered callback routine will be called when the HID value of any element of type kIOHIDElementTypeInput changes for all matching HID devices.

Note: To unregister pass `NULL` for the callback.

Note: The HID Manager must be scheduled with a run loop for HID Manager callbacks to be dispatched.

See Listing 2-6 (page 19) for more information.

Listing 2-9 Registering for an input value callback

```
IOHIDManagerRegisterInputValueCallback(managerRef, Handle_IOHIDInputValueCallback,
 context);
```

This routine will be called when an input value changes for any input element for all matching devices.

Listing 2-10 Example input value callback routine

```
static void Handle_IOHIDInputValueCallback(
                void *          inContext,      // context from
IOHIDManagerRegisterInputValueCallback
                IOReturn        inResult,       // completion result for the
input value operation
                void *          inSender,       // the IOHIDManagerRef
                IOHIDValueRef   inIOHIDValueRef // the new element value
) {
    printf("%s(context: %p, result: %p, sender: %p, value: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender, (void*)
inIOHIDValueRef);
}   // Handle_IOHIDInputValueCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Note: HID values are documented in the "HID Value Functions" (page 45) section.

If only notifications from specific devices are of interest, then the `IOHIDDeviceRegisterInputValueCallback` function (described in Listing 2-20 (page 27)) should be used.

For value changes on specific HID elements, the HID queue functions (described in "HID Queue Functions" (page 38)) should be used.

To receive notifications when HID reports are received from a HID device, the `IOHIDDeviceGetReport` or `IOHIDDeviceGetReportWithCallback` functions (described in Listing 2-28 (page 33)) may be used.

```
void IOHIDManagerSetInputValueMatching(
        IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
        CFDictionaryRef inMatchingDictRef); // single dictionary containing
                                            // element matching criteria.

// Sets multiple matching criteria (array of dictionaries)
// for the input value callback.
void IOHIDManagerSetInputValueMatchingMultiple(
        IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
        CFArrayRef      inCFArrayRef);      // array of dictionaries containing
                                            // element matching criteria.
```

Note: The default element criteria is to match all elements. Specific matching criteria can be reset to this default by passing `NULL` to ether of the above APIs.

Note: The `IOHIDManagerSetInputValueMatching`, `IOHIDManagerSetInputValueMatchingMultiple`, `IOHIDDeviceSetInputValueMatching`, and `IOHIDDeviceSetInputValueMatchingMultiple` APIs (documented below) override each other. The last one called has precedence.

The matching keys for HID elements are prefixed by `kIOHIDElement`. They are declared in `<IOKit/hid/IOHIDKeys.h>`. See Listing 2-31 (page 34) for more information.

The `IOHIDManagerGetProperty` and `IOHIDManagerSetProperty` functions are available to access the HID Manager's properties:

```
// Obtains a property of a HIDManagerRef
CFTypeRef IOHIDManagerGetProperty(
            IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
            CFStringRef     inKeyCFStringRef);  // CFStringRef for the key

// Sets a property for a HIDManagerRef
void IOHIDManagerSetProperty(
            IOHIDManagerRef inIOHIDManagerRef,  // HID Manager reference
            CFStringRef     inKeyCFStringRef,   // CFStringRef for the key
          CFTypeRef      inValueCFTypeRef);  // the HID value for the property
```

Currently there are not any default HID Manager properties set by the system. However since HID Manager properties are propagated to all HID devices as they are enumerated (matched) this might be a convient way to set default HID device property values.

Note: Currently all HID Manager, device, and element properties are lost when the HID Manager reference that they are associated with is closed. Developers should save and restore any values that they want to persist outside that scope.

Listing 2-11 Accessing HID Manager properties

```
CFTypeRef tCFTypeRef = IOHIDManagerGetProperty(managerRef, key);
IOHIDManagerSetProperty(managerRef, key, tCFTypeRef);
```

To determine what devices match the current matching criteria use IOHIDManagerCopyDevices:

```
CFSetRef IOHIDManagerCopyDevices(IOHIDManagerRef inIOHIDManagerRef);    // HID
 Manager reference
```

The parameter is a HID Manager reference. This call returns a Core Foundation set (`CFSetRef`) of `IOHIDDeviceRef` objects.

Listing 2-12 Getting the set of matching HID device references

```
CFSetRef tCFSetRef = IOHIDManagerCopyDevices(managerRef);
```

The HID device references in the returned set can be obtained by using the `CFSetGetValues` function or iterated over by using the `CFSetApplyFunction` function.

HID Device Functions

A `CFTypeRef` object can be verified to be a HID device reference by comparing its Core Foundation type against `IOHIDDeviceGetTypeID`:

Listing 2-13 Validating a HID device reference

```
    if (CFGetTypeID(tCFTypeRef) == IOHIDDeviceGetTypeID()) {
        // this is a valid HID device reference
    }
```

Once you have a valid HID device reference the `IOHIDDeviceGetProperty` function can be used to access its properties (manufacturer, vendor, product IDs, and so on) using the HID device keys defined in `<IOKit/HID/IOHIDKeys.h>`. See Listing 2-2 (page 14) for more information.

Listing 2-14 Examples of getting HID device properties

```
// Get a HID device's transport (string)
CFStringRef IOHIDDevice_GetTransport(IOHIDDeviceRef inIOHIDDeviceRef)
{
    return IOHIDDeviceGetProperty(inIOHIDDeviceRef, CFSTR(kIOHIDTransportKey));
}

// function to get a long device property
// returns FALSE if the property isn't found or can't be converted to a long
static Boolean IOHIDDevice_GetLongProperty(
    IOHIDDeviceRef inDeviceRef,     // the HID device reference
    CFStringRef inKey,              // the kIOHIDDevice key (as a CFString)
    long * outValue)                // address where to return the output value
{
    Boolean result = FALSE;

    CFTypeRef tCFTypeRef = IOHIDDeviceGetProperty(inDeviceRef, inKey);
    if (tCFTypeRef) {
        // if this is a number
        if (CFNumberGetTypeID() == CFGetTypeID(tCFTypeRef)) {
            // get its value
            result = CFNumberGetValue((CFNumberRef) tCFTypeRef,
kCFNumberSInt32Type, outValue);
        }
    }
    return result;
```

```
}   // IOHIDDevice_GetLongProperty

// Get a HID device's vendor ID (long)
long IOHIDDevice_GetVendorID(IOHIDDeviceRef inIOHIDDeviceRef)
{
    long result = 0;
   (void) IOHIDDevice_GetLongProperty(inIOHIDDeviceRef, CFSTR(kIOHIDVendorIDKey),
 &result);
    return result;
} // IOHIDDevice_GetVendorID

// Get a HID device's product ID (long)
long IOHIDDevice_GetProductID(IOHIDDeviceRef inIOHIDDeviceRef)
{
    long result = 0;
    (void) IOHIDDevice_GetLongProperty(inIOHIDDeviceRef,
CFSTR(kIOHIDProductIDKey), &result);
    return result;
} // IOHIDDevice_GetProductID
```

Determining Suitability

There is a convenience function that will scan a device's application collection elements to determine if the device conforms to a specified usage page and usage:

```
Boolean IOHIDDeviceConformsTo(IOHIDDeviceRef   inIOHIDDeviceRef,  //
IOHIDDeviceRef for the HID device
                              uint32_t        inUsagePage,      // the usage
 page to test conformance with
                              uint32_t        inUsage);         // the usage
 to test conformance with
```

Some examples of application collection usage pairs are:

- usagePage = kHIDPage_GenericDesktop, usage = kHIDUsage_GD_Mouse
- usagePage = kHIDPage_GenericDesktop, usage = kHIDUsage_GD_Keyboard

Before you can communicate with a HID device it has to be opened; Opened HID device references should be closed when communications are complete. Here are the functions to open and close a HID device reference:

```
IOReturn IOHIDDeviceOpen(IOHIDDeviceRef  inIOHIDDeviceRef, // IOHIDDeviceRef
for the HID device
                         IOOptionBits    inOptions);      // Option bits to be
 sent down to the HID device
IOReturn IOHIDDeviceClose(IOHIDDeviceRef  IOHIDDeviceRef,  // IOHIDDeviceRef
for the HID device
                         IOOptionBits    inOptions);      // Option bits to be
 sent down to the HID device
```

On the `IOHIDDeviceOpen` call developers may pass `kIOHIDOptionsTypeNone` or `kIOHIDOptionsTypeSeizeDevice` option to request exclusive access to the HID device. Both functions return `kIOReturnSuccess` if successful.

Note: As of Leopard, the `kIOHIDOptionsTypeSeizeDevice` option requires root privileges to be used with keyboard devices.

Obtaining Elements for a Device

To obtain the HID elements associated with a specific device use the `IOHIDDeviceCopyMatchingElements` function:

```
// return the HID elements that match the criteria contained in the matching
dictionary
CFArrayRef IOHIDDeviceCopyMatchingElements(
                IOHIDDeviceRef  inIOHIDDeviceRef,       // IOHIDDeviceRef for
the HID device
              CFDictionaryRef inMatchingCFDictRef,    // the matching dictionary
                IOOptionBits    inOptions);             // Option bits
```

The first parameter is the HID Manager reference. The second parameter is a matching dictionary (which may be `NULL` to return all elements). The third parameter contains any option bits (currently unused, pass `kIOHIDOptionsTypeNone`). This API returns a `CFArrayRef` object containing `IOHIDElementRef` objects. Developers may then use `CFArrayGetValueAtIndex` function to retrieve a specific `IOHIDElementRef` object, `CFArrayGetValues` to retrieve all `IOHIDElementRef` objects or `CFSetApplyFunction` to iterate all `IOHIDElementRef` objects in this array.

The matching keys for HID elements are prefixed by `kIOHIDElement`. They are declared in `<IOKit/hid/IOHIDKeys.h>`. See Listing 2-31 (page 34) for more information.

Listing 2-15 IOHIDDeviceCopyMatchingElements examples

```
// to return all elements for a device
CFArrayRef elementCFArrayRef = IOHIDDeviceCopyMatchingElements(deviceRef, NULL,
 kIOHIDOptionsTypeNone);

// to return all elements with usage page keyboard

// create a dictionary to add element properties to
CFMutableDictionaryRef tCFDictRef = CFDictionaryCreateMutable(
    kCFAllocatorDefault, 0, &kCFTypeDictionaryKeyCallBacks,
&kCFTypeDictionaryValueCallBacks);
if (tCFDictRef) {
    // Add key for element usage page to matching dictionary
    int usagePage = kHIDUsage_GD_Keyboard;
    CFNumberRef pageCFNumberRef = CFNumberCreate(kCFAllocatorDefault,
kCFNumberIntType, &usagePage);
    if (pageCFNumberRef) {
        CFDictionarySetValue(tCFDictRef, CFSTR(kIOHIDElementUsagePageKey),
pageCFNumberRef);
        CFRelease(pageCFNumberRef);
```

```
    } else {
        fprintf(stderr, "%s: CFNumberCreate(usage page) failed.",
__PRETTY_FUNCTION__);
    }
} else {
    fprintf(stderr, "%s: CFDictionaryCreateMutable failed.", __PRETTY_FUNCTION__);
}

if (tCFDictRef) {
    CFArrayRef elementCFArrayRef = IOHIDDeviceCopyMatchingElements(
        deviceRef, tCFDictRef, kIOHIDOptionsTypeNone);
    CFRelease(tCFDictRef);
}
```

Registering for Unplug Notifications

Callbacks can be registered that will be called when a HID device is unplugged, when input values change, when input reports are received, or when asynchronous get and set value and report functions complete. (These callbacks are documented below.) Before these HID device callbacks are dispatched, however, the HID device must be scheduled with a run loop.

Note: If a HID Manager is scheduled with a run loop, then by default, when new devices are matched by that HID Manager, they are automatically scheduled with the same run loop, in which case this additional step is unnecessary.

Listing 2-16 Scheduling a HID device with a run loop

```
    IOHIDDeviceScheduleWithRunLoop(inIOHIDDeviceRef, CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode);
```

There is a corresponding function to unschedule a HID device from a run loop:

Listing 2-17 Unscheduling a HID device from a run loop

```
IOHIDDeviceUnscheduleFromRunLoop(deviceRef, CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode);
```

To register a routine to be called when a HID device is removed:

Listing 2-18 Registering a HID device removal callback routine

```
IOHIDDeviceRegisterRemovalCallback(deviceRef, Handle_IOHIDDeviceRemovalCallback,
 context);
```

Note: To unregister pass `NULL` for the callback.

Listing 2-19 HID device removal callback routine

```
static void Handle_IOHIDDeviceRemovalCallback(
                void *      inContext,  // context from
IOHIDDeviceRegisterRemovalCallback
                IOReturn    inResult,   // the result of the removal
                void *      inSender    // IOHIDDeviceRef for the HID device
being removed
) {
    printf("%s(context: %p, result: %p, sender: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender);
}   // Handle_IOHIDDeviceRemovalCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Registering for Value Change Notifications

To register a routine to be called when an input value is changed by a HID device:

Listing 2-20 Registering a HID device input value callback routine

```
IOHIDDeviceRegisterInputValueCallback(deviceRef,
Handle_IOHIDDeviceInputValueCallback, context);
```

The first parameter is a HID device reference. The second parameter is the callback routine. The third parameter is a user context parameter that is passed to that callback routine.

Note: To unregister pass `NULL` for the callback.

Listing 2-21 HID device input value callback routine

```
static void Handle_IOHIDDeviceInputValueCallback(
                void *          inContext,      // context from
IOHIDDeviceRegisterInputValueCallback
                IOReturn        inResult,       // completion result for the
input value operation
                void *          inSender,       // IOHIDDeviceRef of the device
 this element is from
                IOHIDValueRef   inIOHIDValueRef // the new element value
) {
    printf("%s(context: %p, result: %p, sender: %p, value: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender, (void*)
inIOHIDValueRef);
}   // Handle_IOHIDDeviceInputValueCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Note: HID values are documented in the "HID Value Functions" (page 45) section.

To limit the element value changes reported by this callback to specific HID elements, an element matching dictionary (single criteria) or array of matching dictionaries (multiple criteria) may be set using the `IOHIDDeviceSetInputValueMatching` or `IOHIDDeviceSetInputValueMatchingMultiple` functions.

```
// Sets single element matching criteria (dictionary) for the
// input value callback.
void IOHIDDeviceSetInputValueMatching(
       IOHIDDeviceRef  inIOHIDDeviceRef,    // IOHIDDeviceRef for the HID device
        CFDictionaryRef inMatchingDictRef);  // single dictionary containing
                                             // element matching criteria.

// Sets multiple matching criteria (array of dictionaries) for the
// input value callback.
void IOHIDDeviceSetInputValueMatchingMultiple(
       IOHIDDeviceRef  inIOHIDDeviceRef,   // IOHIDDeviceRef for the HID device
        CFArrayRef      inCFArrayRef);      // array of dictionaries containing
                                            // element matching criteria.
```

Note: The default element criteria is to match all elements. Specific matching criteria can be reset to this default by passing `NULL` to ether of the above APIs.

Note: The `IOHIDDeviceSetInputValueMatching`, `IOHIDDeviceSetInputValueMatchingMultiple`, `IOHIDManagerSetInputValueMatching`, and `IOHIDManagerSetInputValueMatchingMultiple` APIs (documented above) override each other. The last one called has precedence.

The matching keys for HID elements are prefixed by `kIOHIDElement`. They are declared in `<IOKit/hid/IOHIDKeys.h>`. See Listing 2-31 (page 34) for more information.

Registering for Input Report Notifications

To register a routine to be called when an input report is issued by a HID device:

Listing 2-22 Registering a HID device input report callback routine

```
CFIndex reportSize = 64;    // note: this should be greater than or equal to
the size of the report
uint8_t report = malloc(reportSize);
IOHIDDeviceRegisterInputReportCallback(deviceRef,          // IOHIDDeviceRef
for the HID device
                                       report,             // pointer to the
report data (uint8_t's)
                                       reportSize,         // number of bytes
 in the report (CFIndex)
                                       Handle_IOHIDDeviceIOHIDReportCallback,
   // the callback routine
```

```
                                        context);           // context passed
to callback
```

The first parameter is a HID device reference. The second is the address where to store the input report. The third parameter is the address of the callback routine. The last parameter is a user context parameter that is passed to that callback routine.

Note: To unregister pass `NULL` for the callback.

The report buffer should be large enough to store the largest report that can be expected to be received from the HID device. This size can be obtained by passing kIOHIDMaxInputReportSizeKey as the key to IOHIDDeviceGetProperty.

Listing 2-23 HID device input report callback routine

```
static void Handle_IOHIDDeviceIOHIDReportCallback(
                void *          inContext,          // context from
IOHIDDeviceRegisterInputReportCallback
                IOReturn        inResult,           // completion result for
the input report operation
                void *          inSender,           // IOHIDDeviceRef of the
device this report is from
                IOHIDReportType inType,             // the report type
                uint32_t        inReportID,         // the report ID
                uint8_t *       inReport,           // pointer to the report
data
                CFIndex         InReportLength)     // the actual size of the
input report
{
    printf("%s(context: %p, result: %p, sender: %p," \
             "type: %d, id: %p, report: %p, length: %d).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender,
        (long) inType, inReportID, inReport, inReportLength);
}   // Handle_IOHIDDeviceIOHIDReportCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Note: The layout and the size of the report data is device specific and requires advanced knowledge of how elements are bundled into reports. While this knowledge is available by parsing HID device descriptors, parsing the descriptors also requires advanced knowledge. A higher level abstraction that doesn't require as much advanced knowledge is the HID transactions APIs described in the "HID Transaction Functions" (page 41) section.

Getting and Setting Output or Feature Values

To set the HID value of a single output or feature type element the `IOHIDDeviceSetValue` (synchronous) or `IOHIDDeviceSetValueWithCallback` (asynchronous) functions may be used. (to set multiple values consider using reports or transactions):

```
// synchronous
IOReturn  tIOReturn = IOHIDDeviceSetValue(
                           deviceRef,      // IOHIDDeviceRef for the HID device
                            elementRef,     // IOHIDElementRef for the HID
element
                          valueRef);     // IOHIDValueRef for the HID element's
 new value

// asynchronous
IOReturn  tIOReturn = IOHIDDeviceSetValueWithCallback(
                            deviceRef,          // IOHIDDeviceRef for the HID
device
                            elementRef,         // IOHIDElementRef for the HID
 element
                            valueRef,           // IOHIDValueRef for the HID
element's new value
                            tCFTimeInterval,    // timeout duration
                          Handle_IOHIDDeviceSetValueCallback,  // the callback
 routine
                            context);           // context passed to callback
```

The first parameter is a HID device reference. The second is a HID element reference. The third parameter is a HID value reference. For the asynchronous version, the fourth parameter is a timeout, the fifth parameter is the callback routine, and the last parameter is a context pointer that is passed to that callback routine.

Listing 2-24 HID device set value callback routine

```
static void Handle_IOHIDDeviceSetValueCallback(
                void *          inContext,          // context from
IOHIDDeviceSetValueWithCallback
                IOReturn        inResult,           // completion result for
the set value operation
                void *          inSender,           // IOHIDDeviceRef of the
device
                IOHIDValueRef   inIOHIDValueRef)    // the HID element value
{
    printf("%s(context: %p, result: %p, sender: %p, value: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender,
inIOHIDValueRef);
}   // Handle_IOHIDDeviceSetValueCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Note: HID values are documented in the "HID Value Functions" (page 45) section.

To get the HID value of a single element the `IOHIDDeviceGetValue` (synchronous) or `IOHIDDeviceGetValueWithCallback` (asynchronous) functions may be used. (To get multiple values consider using reports or transactions.) For input type elements the synchronous function returns immediately; for feature type elements it will block until the get value report has been issued to the HID device.

```
// synchronous
IOReturn  tIOReturn = IOHIDDeviceGetValue(
                            deviceRef,  // IOHIDDeviceRef for the HID device
                            elementRef, // IOHIDElementRef for the HID element
                            valueRef);  // IOHIDValueRef for the HID element's
 new value

// asynchronous
IOReturn  tIOReturn = IOHIDDeviceGetValueWithCallback(
                            deviceRef,          // IOHIDDeviceRef for the HID
device
                            elementRef,         // IOHIDElementRef for the HID
 element
                            valueRef,           // IOHIDValueRef for the HID
element's new value
                            tCFTimeInterval,    // timeout duration
                           Handle_IOHIDDeviceGetValueCallback,  // the callback
 routine
                            context);           // context passed to callback
```

For both of these functions the first parameter is a HID device reference. The second is a HID element reference. The third parameter is a HID value reference. For the asynchronous version, the fourth parameter is a timeout, the fifth parameter is a callback routine, and the last parameter is a context pointer that is passed to that callback routine.

Listing 2-25 HID device get value callback routine

```
static void Handle_IOHIDDeviceGetValueCallback(
                void *          inContext,          // context from
IOHIDDeviceGetValueWithCallback
                IOReturn        inResult,           // completion result for
the get value operation
                void *          inSender,           // IOHIDDeviceRef of the
device
                IOHIDValueRef   inIOHIDValueRef)    // the HID element value
{
    printf("%s(context: %p, result: %p, sender: %p, value: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender,
inIOHIDValueRef);
}   // Handle_IOHIDDeviceGetValueCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Note: HID values are documented in the "HID Value Functions" (page 45) section.

Getting and Setting HID Reports

USB data is transferred to and from HID devices packetized into reports. These reports consist of one or more element fields usually contained in a hierarchy of collections. Developers who understand how elements are packaged into reports can use the `IOHIDDeviceGetReport`, `IOHIDDeviceGetReportWithCallback`,

`IOHIDDeviceSetReport`, and `IOHIDDeviceSetReportWithCallback` functions to talk directly with HID devices. Developers unfamiliar with how HID reports are constructed may use the HID transaction functions. See "HID Transaction Functions" (page 41) for more information.

To send a report to a HID device the IOHIDDeviceSetReport (synchronous) or IOHIDDeviceSetReportWithCallback (asynchronous) functions should be used:

Listing 2-26 Sending a HID Report

```
CFIndex reportSize = 64;
uint8_t report = malloc(reportSize);

// synchronous
IOReturn  tIOReturn = IOHIDDeviceSetReport(
                            deviceRef,          // IOHIDDeviceRef for the HID
device
                          tIOHIDReportType,   // IOHIDReportType for the report
                            reportID,           // CFIndex for the report ID
                            report,             // address of report buffer
                            reportLength);      // length of the report

// asynchronous
IOReturn  tIOReturn = IOHIDDeviceSetReportWithCallback(
                            deviceRef,          // IOHIDDeviceRef for the HID
device
                          tIOHIDReportType,   // IOHIDReportType for the report
                            reportID,           // CFIndex for the report ID
                            report,             // address of report buffer
                            reportLength,       // length of the report
                            tCFTimeInterval,    // timeout duration
                          Handle_IOHIDDeviceSetReportCallback,  // the callback
 routine
                            context);           // context passed to callback
```

For both of these functions the first parameter is a HID device reference. The second parameter is an `IOHIDReportType` object for the report. The third parameter is the report ID. The fourth parameter is the address of the report buffer. The fifth parameter is the size of the report being sent. For the asynchronous version, the sixth parameter is a timeout, the seventh parameter is a callback routine, and the last parameter is a context pointer that is passed to that callback routine:

Listing 2-27 HID device set report callback routine

```
static void Handle_IOHIDDeviceSetReportCallback(
                void *          inContext,          // context from
IOHIDDeviceSetReportWithCallback
                IOReturn        inResult,           // completion result for
the set value operation
                void *          inSender,           // IOHIDDeviceRef of the
device this report is from
                IOHIDReportType inIOHIDReportType,  // the report type
                uint32_t        inReportID,         // the report ID
                uint8_t*        inReport,          // the address of the report
                CFIndex         inReportLength)     // the length of the report
{
```

```
    printf("%s(context: %p, result: %p, sender: %p, "  \
            "type: %d, id: %d, report: %p, length: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender,
        inIOHIDReportType, inReportID, inReport, inReportLength);
}   // Handle_IOHIDDeviceSetReportCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

To request a report from a HID device, you should use the `IOHIDDeviceGetReport` (synchronous) or `IOHIDDeviceGetReportWithCallback` (asynchronous) functions as shown below:

Listing 2-28 IOHIDDeviceGetReport and IOHIDDeviceGetReportWithCallback

```
// synchronous
IOReturn  tIOReturn = IOHIDDeviceGetReport(
                            deviceRef,          // IOHIDDeviceRef for the HID
device
                          tIOHIDReportType,   // IOHIDReportType for the report
                            reportID,           // CFIndex for the report ID
                            report,             // address of report buffer
                            &reportSize);       // address of length of the
report

// asynchronous
IOReturn  tIOReturn = IOHIDDeviceGetReportWithCallback(
                            deviceRef,          // IOHIDDeviceRef for the HID
device
                          tIOHIDReportType,   // IOHIDReportType for the report
                            reportID,           // CFIndex for the report ID
                            report,             // address of report buffer
                            &reportSize,        // address of length of the
report
                            tCFTimeInterval,    // timeout duration
                          Handle_IOHIDDeviceGetReportCallback,  // the callback
 routine
                            context);           // context passed to callback
```

For both of these functions, the first parameter is a HID device reference. The second is an `IOHIDReportType` for the report. The third parameter is the report ID. The fourth parameter is the address of the report buffer. The fifth parameter should be the address of a `CFIndex` variable. Initially, you should set the value of this `CFIndex` variable to be the size of the report you are requesting. On return, the new value in that variable is the size of the returned report. For the asynchronous version, the sixth parameter is a timeout, the seventh parameter is a callback routine, and the last parameter is a context pointer that is passed to the callback routine.

Listing 2-29 HID device get report callback routine

```
static void Handle_IOHIDDeviceGetReportCallback(
                void *          inContext,          // context from
IOHIDDeviceGetReportWithCallback
                IOReturn        inResult,           // completion result for
the get report operation
```

```
                void *          inSender,           // IOHIDDeviceRef of the
device this report is from
                IOHIDReportType inIOHIDReportType,  // the report type
                uint32_t        inReportID,         // the report ID
                uint8_t*        inReport,          // the address of the report
                CFIndex         inReportLength)     // the length of the report
{
    printf("%s(context: %p, result: %p, sender: %p, "  \
            "type: %d, id: %d, report: %p, length: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender,
        inIOHIDReportType, inReportID, inReport, inReportLength);
}   // Handle_IOHIDDeviceGetReportCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Working With HID Elements

A CFTypeRef can be verified to be a HID element reference by comparing its Core Foundation type against IOHIDElementGetTypeID:

Listing 2-30 Validating a HID element reference

```
    if (CFGetTypeID(tCFTypeRef) == IOHIDElementGetTypeID()) {
        // this is a valid HID element reference
    }
```

Once a valid HID element reference is available, the `IOHIDElementGetProperty` function may be used to access its properties (type, usage page and usage, and so on) using the HID element keys defined in `<IOKit/HID/IOHIDKeys.h>`:

Listing 2-31 HID element keys

```
From <IOKit/hid/IOHIDKeys.h>:

#define kIOHIDElementCookieKey                  "ElementCookie"
#define kIOHIDElementTypeKey                    "Type"
#define kIOHIDElementCollectionTypeKey          "CollectionType"
#define kIOHIDElementUsageKey                   "Usage"
#define kIOHIDElementUsagePageKey               "UsagePage"
#define kIOHIDElementMinKey                     "Min"
#define kIOHIDElementMaxKey                     "Max"
#define kIOHIDElementScaledMinKey               "ScaledMin"
#define kIOHIDElementScaledMaxKey               "ScaledMax"
#define kIOHIDElementSizeKey                    "Size"
#define kIOHIDElementReportSizeKey              "ReportSize"
#define kIOHIDElementReportCountKey             "ReportCount"
#define kIOHIDElementReportIDKey                "ReportID"
#define kIOHIDElementIsArrayKey                 "IsArray"
#define kIOHIDElementIsRelativeKey              "IsRelative"
```

```
#define kIOHIDElementIsWrappingKey                  "IsWrapping"
#define kIOHIDElementIsNonLinearKey                 "IsNonLinear"
#define kIOHIDElementHasPreferredStateKey           "HasPreferredState"
#define kIOHIDElementHasNullStateKey                "HasNullState"
#define kIOHIDElementFlagsKey                       "Flags"
#define kIOHIDElementUnitKey                        "Unit"
#define kIOHIDElementUnitExponentKey                "UnitExponent"
#define kIOHIDElementNameKey                        "Name"
#define kIOHIDElementValueLocationKey               "ValueLocation"
#define kIOHIDElementDuplicateIndexKey              "DuplicateIndex"
#define kIOHIDElementParentCollectionKey            "ParentCollection"
#define kIOHIDElementVendorSpecificKey              "VendorSpecific"

#define kIOHIDElementCalibrationMinKey              "CalibrationMin"
#define kIOHIDElementCalibrationMaxKey              "CalibrationMax"
#define kIOHIDElementCalibrationSaturationMinKey    "CalibrationSaturationMin"
#define kIOHIDElementCalibrationSaturationMaxKey    "CalibrationSaturationMax"
#define kIOHIDElementCalibrationDeadZoneMinKey      "CalibrationDeadZoneMin"
#define kIOHIDElementCalibrationDeadZoneMaxKey      "CalibrationDeadZoneMax"
#define kIOHIDElementCalibrationGranularityKey      "CalibrationGranularity"
```

Note: Use the `CFSTR` macro to pass these keys to the get/set property functions as `CFStringRef` pointers.

Important: Convenience functions have been provided to allow developers to access many of these properties directly without having to use intermediary Core Foundation types. See Listing 2-34 (page 36) for more information.

Due to an unintentional implementation detail (bug) these element properties may or may not be accessible via the `IOHIDElementGetProperty` and `IOHIDElementSetProperty` functions. Please use the convenience APIs to access these properties.

All the `kIOHIDElementCalibration***Key` properties are accessible via the `IOHIDElementGetProperty` and `IOHIDElementSetProperty` functions.

Listing 2-32 Passing HID element keys to the Get/Set Property functions

```
IOHIDElementGetProperty(element, CFSTR(kIOHIDElementTypeKey), &tCFNumberRef);
```

Here are two functions that can be used to get or set long properties:

Listing 2-33 Examples of how to get or set long HID element properties

```
static Boolean IOHIDElement_GetLongProperty(
    IOHIDElementRef inElementRef,   // the HID element
    CFStringRef inKey,              // the kIOHIDElement key (as a CFString)
    long * outValue)                // address where to return the output value
{
    Boolean result = FALSE;

    CFTypeRef tCFTypeRef = IOHIDElementGetProperty(inElementRef, inKey);
    if (tCFTypeRef) {
```

```
        // if this is a number
        if (CFNumberGetTypeID() == CFGetTypeID(tCFTypeRef)) {
            // get its value
            result = CFNumberGetValue((CFNumberRef) tCFTypeRef,
kCFNumberSInt32Type, outValue);
        }
    }
    return result;
}

static void IOHIDElement_SetLongProperty(
    IOHIDElementRef inElementRef,   // the HID element
    CFStringRef inKey,              // the kIOHIDElement key (as a CFString)
    long inValue)                   // the long value to be set
{
    CFNumberRef tCFNumberRef = CFNumberCreate(kCFAllocatorDefault,
kCFNumberSInt32Type, &inValue);
    if (tCFNumberRef) {
        IOHIDElementSetProperty(inElementRef, inKey, tCFNumberRef);
        CFRelease(tCFNumberRef);
    }
}

// access the kIOHIDElementVendorSpecificKey if it exists:
long longValue;
if (IOHIDElement_GetLongProperty(elementRef,
CFSTR(kIOHIDElementVendorSpecificKey), &longValue)) {
    printf("Element 0x%08lX has a vendor specific key of value 0x%08lX.\n",
elementRef, longValue);
}
```

There are convenience functions to retrieve many of these element properties directly:

Listing 2-34 HID element property functions

```
// IOHIDElementCookie represent a unique identifier for a HID element within a
 HID device.
IOHIDElementCookie cookie = IOHIDElementGetCookie(elementRef);

// return the collection type:
//  kIOHIDElementTypeInput_Misc         = 1,
//  kIOHIDElementTypeInput_Button       = 2,
//  kIOHIDElementTypeInput_Axis         = 3,
//  kIOHIDElementTypeInput_ScanCodes    = 4,
//  kIOHIDElementTypeOutput             = 129,
//  kIOHIDElementTypeFeature            = 257,
//  kIOHIDElementTypeCollection         = 513
IOHIDElementType tType = IOHIDElementGetType(elementRef);

// If the HID element type is of type kIOHIDElementTypeCollection then
// the collection type is one of:
//  kIOHIDElementCollectionTypePhysical         = 0x00,
//  kIOHIDElementCollectionTypeApplication      = 0x01,
//  kIOHIDElementCollectionTypeLogical          = 0x02,
//  kIOHIDElementCollectionTypeReport           = 0x03,
```

```
//  kIOHIDElementCollectionTypeNamedArray       = 0x04,
//  kIOHIDElementCollectionTypeUsageSwitch      = 0x05,
//  kIOHIDElementCollectionTypeUsageModifier    = 0x06
IOHIDElementCollectionType collectionType =
IOHIDElementGetCollectionType(elementRef);

// usage and usage pages are defined on the USB website at: <http://www.usb.org>
uint32_t page = IOHIDElementGetUsagePage(elementRef);
uint32_t usage = IOHIDElementGetUsage(elementRef);

// Boolean properties
Boolean isVirtual = IOHIDElementIsVirtual(elementRef);
Boolean isRelative = IOHIDElementIsRelative(elementRef);
Boolean isWrapping = IOHIDElementIsWrapping(elementRef);
Boolean isArray = IOHIDElementIsArray(elementRef);
Boolean isNonLinear = IOHIDElementIsNonLinear(elementRef);
Boolean hasPreferred = IOHIDElementHasPreferredState(elementRef);
Boolean hasNullState = IOHIDElementHasNullState(elementRef);

// the HID element name
CFStringRef name = IOHIDElementGetName(elementRef);

// element report information
uint32_t reportID = IOHIDElementGetReportID(elementRef);
uint32_t reportSize = IOHIDElementGetReportSize(elementRef);
uint32_t reportCount = IOHIDElementGetReportCount(elementRef);

// element unit & exponent
uint32_t unit = IOHIDElementGetUnit(elementRef);
uint32_t unitExp = IOHIDElementGetUnitExponent(elementRef);

// logical & physical minimums & maximums
CFIndex logicalMin = IOHIDElementGetLogicalMin(elementRef);
CFIndex logicalMax = IOHIDElementGetLogicalMax(elementRef);
CFIndex physicalMin = IOHIDElementGetPhysicalMin(elementRef);
CFIndex physicalMax = IOHIDElementGetPhysicalMax(elementRef);
```

There are also functions to determine the device, parent, and child of a specified HID element:

Listing 2-35 HID element hierarchy functions

```
// return the HID device that a element belongs to
IOHIDDeviceRef deviceRef = IOHIDElementGetDevice(elementRef);

// return the collection element that a HID element belongs to (if any)
IOHIDElementRef elementRef = IOHIDElementGetParent(elementRef);

// return the child elements of a collection element (if any)
CFArrayRef tCFArrayRef = IOHIDElementGetChildren(elementRef);
```

HID Queue Functions

While developers can use the `IOHIDDeviceGetValue` to get the most recent value of a HID element, for some elements this is not sufficient. If it is necessary to keep track of all value changes of a HID element, rather than just the most recent one, developers can create a queue and add the HID elements of interest to it. After doing so, all value change events involving those elements are captured by the HID queue (up to the depth of the HID queue).

HID queue references (`IOHIDQueueRef` objects) are used to communicate with the HID queues. They are created by using the `IOHIDQueueCreate` function:

```
// Create HID queue reference
IOHIDQueueRef IOHIDQueueCreate(
                    CFAllocatorRef  inCFAllocatorRef,   // Allocator to be used
 during creation
                    IOHIDDeviceRef  inIOHIDDeviceRef,   // the HID device to be
 associated with this queue
                     CFIndex         inDepth,            // the maximum number
of values to queue
                     IOOptionBits    inOptions)          // options (currently
reserved)
```

The first parameter is a `CFAllocatorRef` object to be used when allocating the returned `IOHIDQueueRef`. The second parameter is the HID device to be associated with this queue. The third parameter is the maximum depth of the HID queue. The last parameter (`options`) is currently reserved for future use. Developers should pass `kIOHIDOptionsTypeNone` (zero) for this parameter.

There is no `IOHIDQueueDestroy` (or release, free, and so on). Because the HID queue reference is a Core Foundation object reference, `CFRelease` should be used to dispose of it.

A `CFTypeRef` can be verified to be a HID queue reference by comparing its Core Foundation type against `IOHIDQueueGetTypeID`:

Listing 2-36 Validating a HID queue reference

```
    if (CFGetTypeID(tCFTypeRef) == IOHIDQueueGetTypeID()) {
        // this is a valid HID queue reference!
    }
```

Once a HID queue reference has been created, it has to be started before it can be used to access the HID devices associated with it.

```
void IOHIDQueueStart(IOHIDQueueRef inIOHIDQueueRef);
```

The corresponding function to stop a HID queue is:

```
void IOHIDQueueStop(IOHIDQueueRef inIOHIDQueueRef);
```

Note: HID queues have to be stopped before HID elements can be added or removed. Also HID elements can only be added to the HID queue for their device. You can't use a single HID queue for multiple devices.

To determine the HID device associated with a specific HID queue use the `IOHIDQueueGetDevice` function:

```
IOHIDDeviceRef IOHIDQueueGetDevice(IOHIDQueueRef inIOHIDQueueRef);
```

There are accessor function to get and set the HID queue's depth:

```
CFIndex IOHIDQueueGetDepth(IOHIDQueueRef inIOHIDQueueRef);
void IOHIDQueueSetDepth(IOHIDQueueRef inIOHIDQueueRef, CFIndex inDepth);
```

HID elements can be added and removed by using these functions:

```
void IOHIDQueueAddElement(IOHIDQueueRef inIOHIDQueueRef, IOHIDElementRef
inIOHIDElementRef);
void IOHIDQueueRemoveElement(IOHIDQueueRef inIOHIDQueueRef, IOHIDElementRef
inIOHIDElementRef);
```

To determine if a HID element has been added to a HID queue use this function:

```
Boolean IOHIDQueueContainsElement(IOHIDQueueRef inIOHIDQueueRef, IOHIDElementRef
 inIOHIDElementRef);
```

Once a HID queue has been created, HID elements have been added, and the queue has been started, HID values can then be dequeued with one of these functions:

```
IOHIDValueRef IOHIDQueueCopyNextValue(IOHIDQueueRef inIOHIDQueueRef);
IOHIDValueRef IOHIDQueueCopyNextValueWithTimeout(IOHIDQueueRef inIOHIDQueueRef,
 CFTimeInterval inTimeout);
```

Note: The first function is synchronous and will block until there is a HID value available. While this may be desirable when called from a secondary thread blocking as the main thread should always be avoided. So on the main thread developers will most likely want to use the the second function with a zero timeout. This is essentially a method for polling the HID queue without blocking.

Note: Because the HID value is a retained copy, it is up to the caller to release the HID value (using `CFRelease`).

Note: HID values are documented in the "HID Value Functions" (page 45) section.

To avoid polling the HID queue for HID value changes developers can instead register a callback routine:

```
void IOHIDQueueRegisterValueAvailableCallback(
                                IOHIDQueueRef inIOHIDQueueRef,   // reference
to the HID queue
                                IOHIDCallback inCallback,        // address of
 the callback routine
                                void *        inContext);        // context
passed to callback
```

Note: The HID queue must be scheduled with a run loop for this callback routine to be dispatched.

The functions to schedule and unschedule a HID queue from a run loop are:

```
// Schedule a HID queue with a runloop
void IOHIDQueueScheduleWithRunLoop(IOHIDQueueRef       inIOHIDQueueRef,     //
 reference to the HID queue
                                   CFRunLoopRef        inRunLoop,           //
 Run loop to be scheduled with
                                   CFStringRef         inRunLoopMode);      //
 Run loop mode for scheduling

// Unschedule a HID queue from a runloop
void IOHIDQueueUnscheduleFromRunLoop(
                        IOHIDQueueRef   inIOHIDQueueRef,    // reference to the
 HID queue
                        CFRunLoopRef    inRunLoop,          // Run loop to be
unscheduling from
                        CFStringRef     inRunLoopMode);     // Run loop mode
for unscheduling
```

Listing 2-37 Scheduling a HID queue with a run loop

```
IOHIDQueueScheduleWithRunLoop(inIOHIDQueueRef, CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode);
```

Listing 2-38 HID queue value available callback routine

```
static void Handle_ValueAvailableCallback(
                void *   inContext, // context from
IOHIDQueueRegisterValueAvailableCallback
                IOReturn inResult,  // the inResult
                void *   inSender,  // IOHIDQueueRef of the queue
) {
    printf("%s(context: %p, result: %p, sender: %p).\n",
        __PRETTY_FUNCTION__, inContext, (void *) inResult, inSender);
    do {
        IOHIDValueRef valueRef =
IOHIDQueueCopyNextValueWithTimeout((IOHIDQueueRef) inSender, 0.);
        if (!valueRef) break;
        // process the HID value reference
```

```
        .
        .
        .
       CFRelease(valueRef);    // Don't forget to release our HID value reference
    } while (1) ;
}   // Handle_ValueAvailableCallback
```

The `inResult` callback parameter contains the error result of the operation that is calling the callback. You should check the return value, and if it is nonzero, handle the failure accordingly.

Note: This routine is not called every time a new value is added to a queue; It is only called when the HID queue transitions to non-empty. For this reason the HID queue should be emptied (by calling `IOHIDQueueCopyNextValueWithTimeout` until it returns `NULL`) before expecting this routine to be called again.

Note: HID values are documented in the "HID Value Functions" (page 45) section.

HID Transaction Functions

Lower-level APIs such as `IOHIDDeviceGetReport`, `IOHIDDeviceGetReportWithCallback`, `IOHIDDeviceSetReport`, and `IOHIDDeviceSetReportWithCallback` require you to know how HID device descriptors are used in order to define the reports sent to and received from HID devices. See Listing 2-28 (page 33) and Listing 2-26 (page 32) for more information about these functions.

HID transactions are an abstraction layered on top of these lower-level APIs. HID transactions allow you to assemble a transaction, add the relevant values, set default values, and commit the transaction (which forces a report to be sent across the USB bus).

To build a transaction, you first must create a HID transaction reference by calling the `IOHIDTransactionCreate` function:

```
IOHIDTransactionRef IOHIDTransactionCreate(
                        CFAllocatorRef  inCFAllocatorRef,   // Allocator to be
 used during creation
                        IOHIDDeviceRef  inIOHIDDeviceRef,   // the HID device
for this transaction
                        IOHIDTransactionDirectionType   inDirection,  // The
direction: in or out
                      IOOptionBits    inOptions);        // options (currently
 reserved)
```

The first parameter is a `CFAllocatorRef` allocator to be used when allocating the returned `IOHIDTransactionRef` object. The second parameter is the HID device to be associated with this transaction. The third parameter is the direction for the transfer (`kIOHIDTransactionDirectionTypeInput` or `kIOHIDTransactionDirectionTypeOutput`). The last parameter (`options`) is currently reserved for future use. Developers should pass `kIOHIDOptionsTypeNone` (zero) for this parameter.

Note: HID transaction references can be used to send and receive multiple element values. The direction used should represent the type of HID elements that you are adding to the transaction.

There is no `IOHIDTransactionDestroy` (or release, free, and so on). Because the HID transaction reference is a Core Foundation object reference, you should call `CFRelease` to dispose of it.

A `CFTypeRef` object can be verified to be a HID transaction reference by comparing its Core Foundation type against the return value of the `IOHIDTransactionGetTypeID` function:

Listing 2-39 Validating a HID transaction reference

```
if (CFGetTypeID(tCFTypeRef) == IOHIDTransactionGetTypeID()) {
    // this is a valid HID transaction reference!
}
```

There are convenience functions to get the HID device associated with a HID transaction and to get or set the direction for a HID transaction:

```
// Obtain the HID device associated with a transaction
IOHIDDeviceRef IOHIDTransactionGetDevice(
                                    IOHIDTransactionRef inIOHIDTransactionRef);
 // HID transaction reference

// Obtain the direction of the transaction.
IOHIDTransactionDirectionType IOHIDTransactionGetDirection(
                                     IOHIDTransactionRef IOHIDTransactionRef);
  // HID transaction reference

// Sets the direction of the transaction
void IOHIDTransactionSetDirection(IOHIDTransactionRef IOHIDTransactionRef,
  // HID transaction reference
                                    IOHIDTransactionDirectionType direction);
  // The direction: in or out
```

Note: The `IOHIDTransactionSetDirection` function is useful for manipulating bi-direction (feature) elements such that you can set or get element values without having to create an additional transaction object.

Once a HID transaction has been created then the HID elements associated with it may be added by using the `IOHIDTransactionAddElement` function:

```
void IOHIDTransactionAddElement(
            IOHIDTransactionRef inIOHIDTransactionRef,  // HID transaction
reference
            IOHIDElementRef     inIOHIDElementRef);     // the HID element to
associate with this transaction
```

Important: To minimize device traffic, you should only add HID elements that share a common report type and id.

HID Elements may be removed from a HID transaction by using the IOHIDTransactionRemoveElement function:

```
void IOHIDTransactionRemoveElement(
            IOHIDTransactionRef inIOHIDTransactionRef,  // HID transaction
reference
            IOHIDElementRef     inIOHIDElementRef);     // the HID element to
associate with this transaction
```

To determine if a HID element is currently associated with a HID transaction the IOHIDTransactionContainsElement function may be used:

```
// Queries the transaction to determine if elemement has been added.
Boolean IOHIDTransactionContainsElement(
                  IOHIDTransactionRef inIOHIDTransactionRef, // HID transaction
 reference
                   IOHIDElementRef     inIOHIDElementRef);    // the HID element
 to test for
```

To set the HID values associated with the HID elements in a HID transaction use the IOHIDTransactionSetValue functions:

```
void IOHIDTransactionSetValue(IOHIDTransactionRef  inIOHIDTransactionRef,   //
 HID transaction reference
                               IOHIDElementRef     inIOHIDElementRef,      //
 the HID element
                               IOHIDValueRef       inIOHIDValueRef,        //
 the HID element value
                               IOOptionBits        inOptions);             //
 options
```

The HID value set is pended until the transaction is committed. This value is only used if the transaction direction is `kIOHIDTransactionDirectionTypeOutput`. Use the `kIOHIDTransactionOptionDefaultOutputValue` option to set the default element values for transactions.

To retrieve the HID values associated with the HID elements in a HID transaction, developers may use the IOHIDTransactionGetValue function:

```
// Obtains the HID value for a transaction element.
IOHIDValueRef IOHIDTransactionGetValue(
                               IOHIDTransactionRef inIOHIDTransactionRef,  //
 HID transaction reference
                               IOHIDElementRef     inIOHIDElementRef,      //
 the HID element
                               IOOptionBits        inOptions);             //
 options
```

If the HID transaction direction is kIOHIDTransactionDirectionTypeInput the HID value represents what was obtained from the HID device from the HID transaction. Otherwise, if the transaction direction is kIOHIDTransactionDirectionTypeOutput the HID value represents the pending value to be sent to the HID device. Use the kIOHIDTransactionOptionDefaultOutputValue option to get the default HID value associated with the HID elements of a HID transaction.

The values for HID elements associated with a HID transaction can be reset to their default values by using the IOHIDTransactionClear function:

```
// Clears element transaction values.
void IOHIDTransactionClear(IOHIDTransactionRef inIOHIDTransactionRef);  // HID
 transaction reference
```

Once all the appropriate HID elements have been added to a HID transaction (and values set for output transactions) then in order to cause the actual bus transaction to occur they should be committed by using one of the two following functions:

```
// Synchronously commits element transaction to the HID device.
IOReturn IOHIDTransactionCommit(IOHIDTransactionRef inIOHIDTransactionRef); //
 HID transaction reference

// Asynchronously commits element transaction to the HID device.
IOReturn IOHIDTransactionCommitWithCallback(
                        IOHIDTransactionRef inIOHIDTransactionRef,  // HID
transaction reference
                        CFTimeInterval      inTimeout,              // timeout
 duration
                        IOHIDCallback       inCallback,             // address
 of the callback routine
                        void *              inContext);             // Pointer
 to be passed to the callback
```

For both functions, the first parameter is the HID transaction reference to be committed. For the asynchronous function, the second parameter is a timeout, the third parameter is a callback routine (pass `NULL` if you want synchronous behavior with a timeout), and the last parameter is a context pointer that is passed to the callback routine.

Note: If the direction is set to `kIOHIDTransactionDirectionTypeOutput`, default element values are used if per-transaction element values are not set. If neither a default value nor a per-transaction value is set, that element is omitted from the commit. After a transaction is committed, the per-transaction element values are cleared, but the default values are preserved.

Note: It is possible for elements from different reports to be present in a given transaction, causing a commit to transcend multiple reports. Keep this in mind when setting a appropriate timeout.

Note: The HID transaction must be scheduled with a run loop In order for the callback routine to be dispatched.

The functions to schedule and unschedule a HID transaction from a run loop are:

```
// Schedule a HID transaction with a runloop
void IOHIDTransactionScheduleWithRunLoop(
        IOHIDTransactionRef inIOHIDTransactionRef,  // reference to the HID
transaction
        CFRunLoopRef        inRunLoop,              // Run loop to be scheduled
 with
        CFStringRef         inRunLoopMode);         // Run loop mode for
scheduling

// Unschedule a HID transaction from a runloop
void IOHIDTransactionUnscheduleFromRunLoop(
        IOHIDTransactionRef inIOHIDTransactionRef,  // reference to the HID
transaction
       CFRunLoopRef       inRunLoop,             // Run loop to be unscheduling
 from
        CFStringRef         inRunLoopMode);         // Run loop mode for
unscheduling
```

Listing 2-40 Scheduling a HID transaction with a run loop

```
IOHIDTransactionScheduleWithRunLoop(inIOHIDTransactionRef, CFRunLoopGetCurrent(),
 kCFRunLoopDefaultMode);
```

HID Value Functions

HID value references are used by the HID Manager and HID device input value callbacks, the HID device get and set value functions and callbacks, and the HID queue copy value functions (with or without timeout). Three functions are available for creating HID value references:

```
IOHIDValueRef IOHIDValueCreateWithIntegerValue(
            CFAllocatorRef  inCFAllocatorRef,   // Allocator to be used during
 creation
```

```
            IOHIDElementRef inIOHIDElementRef,  // the HID element to be
associated with this value
            uint64_t        inTimeStamp,        // OS AbsoluteTime
            CFIndex         inValue);           // the integer (32-bit) value
used to create this HID value

IOHIDValueRef IOHIDValueCreateWithBytes(
            CFAllocatorRef  inCFAllocatorRef,   // Allocator to be used during
 creation
            IOHIDElementRef inIOHIDElementRef,  // the HID element to be
associated with this value
            uint64_t        inTimeStamp,        // OS AbsoluteTime
            const uint8_t * inBytes,            // a pointer to the data used
to create this HID value
            CFIndex         inLength);          // the length of the data used
 to create this HID value

IOHIDValueRef IOHIDValueCreateWithBytesNoCopy(
            CFAllocatorRef  inCFAllocatorRef,   // Allocator to be used during
 creation
            IOHIDElementRef inIOHIDElementRef,  // the HID element to be
associated with this value
            uint64_t        inTimeStamp,        // OS AbsoluteTime
            const uint8_t * inBytes,            // a pointer to the data used
to create this HID value
            CFIndex         inLength);          // the length of the data used
 to create this HID value
```

For all three of these functions the first parameter is a `CFAllocatorRef` object to be used when allocating the returned `IOHIDValueRef`. The second parameter is the HID element to be associated with this value. The third parameter is a time stamp. For the `IOHIDValueCreateWithIntegerValue` function, the last parameter is a `CFIndex` value. For the last two functions, the fourth parameter is a pointer to the data, and the last parameter is the length of the data.

Note: For all three of these functions the `timeStamp` value should represent an OS `AbsoluteTime` value, not a `CFAbsoluteTime` value. See `<mach/mach_time.h>` for details.

Note: For the `IOHIDValueCreateWithBytesNoCopy` function the data is expected to exist until the HID value reference is released. Any attempt to access the data after it has been released may result in a crash.

A `CFTypeRef` can be verified to be a HID value reference by comparing its Core Foundation type against `IOHIDValueGetTypeID`:

Listing 2-41 Validating a HID value reference

```
if (CFGetTypeID(tCFTypeRef) == IOHIDValueGetTypeID()) {
    // this is a valid HID value reference!
}
```

Convenience functions are provided to access the HID element, time stamp, and integer values associated with HID value objects:

```
// Returns the HID element value associated with this HID value reference.
IOHIDElementRef IOHIDValueGetElement(IOHIDValueRef inIOHIDValueRef);

// Returns the timestamp value associated with this HID value reference.
uint64_t IOHIDValueGetTimeStamp(IOHIDValueRef inIOHIDValueRef);

// Returns an integer representation for this HID value reference.
CFIndex IOHIDValueGetIntegerValue(IOHIDValueRef inIOHIDValueRef);
```

Additional functions are provided to access the data and the length of the data associated with a HID value object:

```
// Returns the size, in bytes, of the data associated with this HID value
reference.
CFIndex IOHIDValueGetLength(IOHIDValueRef inIOHIDValueRef);

// Returns a byte pointer to the data associated with this HID value reference.
const uint8_t * IOHIDValueGetBytePtr(IOHIDValueRef inIOHIDValueRef)
```

One additional function exists to return a scaled representation of a HID value object:

```
// return the scaled value of a HID value reference
double_t IOHIDValueGetScaledValue(IOHIDValueRef inIOHIDValueRef,
IOHIDValueScaleType inType);
```

There are currently two types of scaling that can be applied:

- kIOHIDValueScaleTypePhysical: Scales values using the physical bounds of the HID element.
- kIOHIDValueScaleTypeCalibrated: Scales values using the calibration properties of the HID element.

Note: Currently there are no calibration properties associated with HID elements by default. Developers are expected to set the appropriate calibration properties for all elements that they want to scale using the IOHIDValueGetScaledValue function with the kIOHIDValueScaleTypeCalibrated scale type.

The first two HID element calibration properties define the desired range of the returned scaled value:

`kIOHIDElementCalibrationMinKey`
: The minimum bounds for a calibrated value (default = -1).

`kIOHIDElementCalibrationMaxKey`
: The maximum bounds for a calibrated value (default = +1).

For example, the actual raw range of a HID element might go from 0-255, but the developer might want the scaled value to be returned with a range of -32.0 to +32.0. In this example, the min and max calibration values would be set to -32.0 and +32.0, respectively.

The next two HID element calibration properties define the range of expected values:

`kIOHIDElementCalibrationSaturationMinKey`
: The minimum value to be used when calibrating a HID value.

`kIOHIDElementCalibrationSaturationMaxKey`
: The maximum value to be used when calibrating a HID value.

Some HID devices may have elements that can't return the full range of values defined by their logical min and max value limits. For example, the logical values for an element might be defined as ranging from 0 to 255, but the actual device may actually only be able to return values in the range of 5 to 250. This may be caused by digitization errors, mechanical limits on an encoder, and so on. If these calibration properties are set, then logical values within this range are scaled out to the full logical range for the HID device. In this example, the min and max saturation values would be set to 5 and 250, respectively.

The next two HID element calibration properties define the range of a dead zone (if it exists):

`kIOHIDElementCalibrationDeadZoneMinKey`
: The minimum bounds near the midpoint where values are ignored.

`kIOHIDElementCalibrationDeadZoneMaxKey`
: The maximum bounds near the midpoint where values are ignored.

Some HID devices (such as joysticks) have elements that have a mechanical return-to-center feature. Because of mechanical slop, drift, or digitization noise, these elements may not always return the exact same values when the HID element is returned to the center position. For example, an element with a logical range of 0 to 255 might return center values ranging from 124 to 130. If these dead zone properties are set (to 124 and 130, in this case), then any value between these two numbers is returned as the center scaled value (127 in this case).

The last HID element calibration property defines a granularity:

`kIOHIDElementCalibrationGranularityKey`
: The scale or level of detail returned in a calibrated element value.

For example, if the granularity property is set to 0.1, the returned values after calibration are exact multiples of 0.1: { 0.0, 0.1, 0.2, 0.3, 0.4, etc. }.

Listing 2-42 Setting HID element calibration properties

```
static void IOHIDElement_SetDoubleProperty(
                                IOHIDElementRef inElementRef,   // the HID
element
                                CFStringRef     inKey,          // the
kIOHIDElement key (as a CFString)
                                double          inValue)        // the double
value to be set
{
    CFNumberRef tCFNumberRef = CFNumberCreate(kCFAllocatorDefault,
kCFNumberDoubleType, &inValue);
    if (tCFNumberRef) {
        IOHIDElementSetProperty(inElementRef, inKey, tCFNumberRef);
        CFRelease(tCFNumberRef);
    }
}

// These define the range of the returned scaled values
IOHIDElement_SetDoubleProperty(elementRef, CFSTR(kIOHIDElementCalibrationMinKey),
 -32.);
```

```
IOHIDElement_SetDoubleProperty(elementRef, CFSTR(kIOHIDElementCalibrationMaxKey),
 +32.);

// these define the range of values expected from the device (logical values)
IOHIDElement_SetDoubleProperty(elementRef,
CFSTR(kIOHIDElementCalibrationSaturationMinKey), 5.);
IOHIDElement_SetDoubleProperty(elementRef,
CFSTR(kIOHIDElementCalibrationSaturationMaxKey), 250.);

// these define the range of the dead zone (logical values)
IOHIDElement_SetDoubleProperty(elementRef,
CFSTR(kIOHIDElementCalibrationDeadZoneMinKey), 124.);
IOHIDElement_SetDoubleProperty(elementRef,
CFSTR(kIOHIDElementCalibrationDeadZoneMaxKey), 130.);

// this defines the granularity of the returned scaled values
IOHIDElement_SetDoubleProperty(elementRef,
CFSTR(kIOHIDElementCalibrationGranularityKey), 0.1);
```

Listing 2-43 Pseudo code for the IOHIDValueGetScaledValue function

```
// first a convenience function to access HID element properties stored as
doubles:
static Boolean IOHIDElement_GetDoubleProperty(
    IOHIDElementRef inElementRef,   // the HID element
    CFStringRef inKey,              // the kIOHIDElement key (as a CFString)
    double * outValue)              // address where to return the output value
{
    Boolean result = FALSE;

    CFTypeRef tCFTypeRef = IOHIDElementGetProperty(inElementRef, inKey);
    if (tCFTypeRef) {
        // if this is a number
        if (CFNumberGetTypeID() == CFGetTypeID(tCFTypeRef)) {
            // get its value
            result = CFNumberGetValue((CFNumberRef) tCFTypeRef,
kCFNumberDoubleType, outValue);
        }
    }
    return result;
}

double_t IOHIDValueGetScaledValue(IOHIDValueRef inValue, IOHIDValueScaleType
inType)
{
    IOHIDElementRef element = IOHIDValueGetElement(inValue);

    double_t logicalValue = IOHIDValueGetIntegerValue(inValue);

    double_t logicalMin = IOHIDElementGetLogicalMin(element);
    double_t logicalMax = IOHIDElementGetLogicalMax(element);

    double_t scaledMin = 0;
    double_t scaledMax = 0;

    double_t granularity = 0.;
```

```
    double_t returnValue = 0.;

    switch (inType) {
        case kIOHIDValueScaleTypeCalibrated: {

            double_t calibrateMin = 0.;
            (void) IOHIDElement_GetDoubleProperty(element,
                        CFSTR(kIOHIDElementCalibrationMinKey), &calibrateMin);
            double_t calibrateMax = 0.;
            (void) IOHIDElement_GetDoubleProperty(element,
                        CFSTR(kIOHIDElementCalibrationMaxKey), &calibrateMax);

            // if there are calibration min/max values...
            if (calibrateMin != calibrateMax) {
                // ...use them...
                scaledMin = calibrateMin;
                scaledMax = calibrateMax;
            } else {
                // ...otherwise use +/- 1.0
                scaledMin = -1.;
                scaledMax = +1.;
            }

            double_t saturationMin = 0.;
            (void) IOHIDElement_GetDoubleProperty(element,
                        CFSTR(kIOHIDElementCalibrationSaturationMinKey),
&saturationMin);
            double_t saturationMax = 0.;
            (void) IOHIDElement_GetDoubleProperty(element,
                        CFSTR(kIOHIDElementCalibrationSaturationMaxKey),
&saturationMax);

            // if there are saturation min/max values...
            if (saturationMin != saturationMax) {
                // .. and the logical value is less than the minimum saturated
 value...
                if (logicalValue <= saturationMin) {
                    // ...then return the minimum scaled value
                    return scaledMin;
                } else
                // otherwise if the logical value is greater than the maximum
saturated value...
                if (logicalValue >= saturationMax) {
                    // ...return the maximum scaled value.
                    return scaledMax;
                } else
                // otherwise use the min/max saturated values for the logical
min/max
                {
                    logicalMin = saturationMin;
                    logicalMax = saturationMax;
                }
            }

            double_t deadzoneMin = 0.;
            (void) IOHIDElement_GetDoubleProperty(element,
```

```
                        CFSTR(kIOHIDElementCalibrationDeadZoneMinKey),
&deadzoneMin);
            double_t deadzoneMax = 0.;
            (void) IOHIDElement_GetDoubleProperty(element,
                        CFSTR(kIOHIDElementCalibrationDeadZoneMaxKey),
&deadzoneMax);

            // if there are deadzone min/max values...
            if (deadzoneMin != deadzoneMax) {
                    double_t scaledMid = (scaledMin + scaledMax) / 2.;

                // if the logical value is less than the deadzone min...
                if (logicalValue < deadzoneMin) {
                    // ...then use the deadzone min as our logical max...
                    logicalMax = deadzoneMin;
                    // ...and the middle of our scaled range as our scaled max.
                    scaledMax = scaledMid;
                } // otherwise if the logical value is greater than the deadzone
 max...
                else if (logicalValue > deadzoneMax) {
                    // ...then use the deadzone max as our logical min...
                    logicalMin = deadzoneMax;
                    // ...and the middle of our scaled range as our scaled min.
                    scaledMin = scaledMid;
                } else {
                    // otherwise return the middle of our scaled range
                    return scaledMid;
                }
            }

            (void) IOHIDElement_GetDoubleProperty(element,
                        CFSTR(kIOHIDElementCalibrationGranularityKey),
&granularity);
            break;
        }
        case kIOHIDValueScaleTypePhysical: {
            scaledMin = IOHIDElementGetPhysicalMin(element);
            scaledMax = IOHIDElementGetPhysicalMax(element);
            break;
        }
        default: {
            return returnValue; // should be 0.0
        }
    }

    double_t logicalRange = logicalMax - logicalMin;
    double_t scaledRange = scaledMax - scaledMin;

    returnValue = ((logicalValue - logicalMin) * scaledRange / logicalRange) +
 scaledMin;

    if (granularity) {
        returnValue = round(returnValue / granularity) * granularity;
    }
    return returnValue;
}
```

Legacy HID Access Overview

This chapter provides step-by-step instructions, including listings of sample code, for accessing a HID class device on older versions of Mac OS X. If you are developing new code for Mac OS X v10.5 or later, you should use the APIs described in "Accessing a HID Device" (page 13).

The sample code shows how to find all HID class devices, how to narrow the search to devices of a specific type, and how to open and communicate with the device.

To search the I/O Registry for a currently available HID class device, create a device interface for the device, and communicate with it, you perform the steps described in the following sections:

1. "HID Access Overview" (page 53)
2. "Setting Up Your Programming Environment" (page 54)
3. "Accessing the I/O Kit Master Port" (page 55)
4. "Finding a HID Class Device" (page 55)

HID Access Overview

The following list of steps briefly describes how to find a HID class device, create a device interface for it, connect to it, and communicate with it. The remainder of this chapter gives a detailed example of this process, including sample code.

1. Find the object that represents the device in the I/O Registry. For an overview of this process, see the discussion of device matching in the chapterDevice Access and the I/O Kit of *Accessing Hardware From Applications*.
2. Create a device interface for the device. For a HID class device, you create a device interface of type `IOHIDDeviceInterface`. This device interface, defined in `IOHIDLib.h`, provides all the functions you need to access and communicate with your device.
3. Open a connection to the device by calling the `open` function of the device interface.
4. Communicate with the device using the functions provided by the HID Manager. For example, to get an element's most recent value, use the `getElementValue` function.

   Functions for queue creation and manipulation are also provided by the HID Manager. For example, to read the next event from a queue, use the `getNextEvent` function; to request notification when a queue is no longer empty, use the function `setEventCallout`.
5. When you are finished with the device, call the `close` function of the device interface.
6. Release the device interface.

Setting Up Your Programming Environment

The sample code in this document is from a Xcode project that builds a command-line tool. This section focuses on the use of Xcode to develop a test function for the device interface functions you'll be using. You can find more detailed documentation for Xcode at http://developer.apple.com/documentation/DeveloperTools/index.html.

To set up Xcode to build the device interface, do the following:

1. Choose "CoreFoundation Tool" (in the Command Line Utility) when creating your project or add `CoreFoundation.framework` to your External Frameworks and Libraries of an existing project.
2. Add `IOKit.framework` and `System.framework` to the External Frameworks and Libraries section of your project. (Click and drag from Finder.)
3. It is recommended that you initially build your project with debugging turned on. To do this in Xcode, open the target list by clicking on the disclosure triangle. Double-click on the only target listed. In the inspector that appears, click on the build tab, then check the checkbox next to 'Generate Debug Symbols'.

Note: This chapter does not contain a complete sample tool. If your goal is to build a sample application to experiment with this code, you should download the companion files associated with this document, which include all of the code samples as a functioning tool. For more information, see "Complete Code Samples" (page 69).

Note: Although this code uses Core Foundation objects extensively, these objects are toll-free bridged to the equivalent Foundation objects (`CFArray` and `NSArray`, for example). For more information on using Core Foundation code in Cocoa applications, read *Carbon-Cocoa Integration Guide*.

Listing 3-1 (page 54) shows the header files you'll need to include for the sample code in this document. Some of these headers include others; a shorter list may be possible. Except for `CoreFoundation.h`, these headers are generally part of `IOKit.framework` or `System.framework`, both of which are located in `/System/Library/Frameworks`.

Listing 3-1 Header files to include for the sample code in this document

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/errno.h>
#include <sysexits.h>
#include <mach/mach.h>
#include <mach/mach_error.h>
#include <IOKit/IOKitLib.h>
#include <IOKit/IOCFPlugIn.h>
#include <IOKit/hid/IOHIDLib.h>
#include <IOKit/hid/IOHIDKeys.h>
#include <CoreFoundation/CoreFoundation.h>
```

Accessing the I/O Kit Master Port

The sample code for working with HID class device interfaces begins with the `MyStartHIDDeviceInterfaceTest` function, shown in Listing 3-2 (page 55). This function establishes a connection to the I/O Kit and then calls functions to search for HID class devices and perform tests on each HID class device found. It accomplishes most of its work by calling the following functions:

- `MyFindHIDDevices`, shown in Listing 3-3 (page 56), sets up a matching dictionary and then searches for matching HID class devices
- `MyTestHIDDevices`, available in the downloadable source package, iterates over the set of matching devices and calls many of the functions in this sample to print property information and create and test a device interface for each device.

Listing 3-2 A function that finds HID class devices in the I/O Registry and performs a test on each device

```
static void MyStartHIDDeviceInterfaceTest(void)
{
    io_iterator_t hidObjectIterator = NULL;
    IOReturn ioReturnValue = kIOReturnSuccess;

    MyFindHIDDevices(kIOMasterPortDefault, &hidObjectIterator);

    if (hidObjectIterator != NULL)
    {
        MyTestHIDDevices(hidObjectIterator);

        //Release iterator. Don't need to release iterator objects.
        IOObjectRelease(hidObjectIterator);
    }
}
```

The `MyStartHIDDeviceInterfaceTest` function releases both the master port it obtained and the iterator returned by the `MyFindHIDDevices` function, but it doesn't release the iterator's objects. They are released in the `MyTestHIDDevices` function after it uses the objects.

Finding a HID Class Device

Before your application can create a device interface to access a device, it must first find the device in the I/O Registry. To do this, you create a matching dictionary specifying the type of device you want to find. Then you get an iterator for the list of devices that match your criteria so you can examine each one.

The following functions demonstrate this process by creating a matching dictionary and searching the I/O Registry for matching devices. If the search is successful, the function `MyFindHIDDevices` (shown in Listing 3-3 (page 56)) returns an iterator for the list of matched devices.

Searching the I/O Registry

The `MyFindHIDDevices` function is the starting point for finding a specific type of device in the I/O Registry. First, it sets up a matching dictionary by passing `kIOHIDDeviceKey` (defined in `IOHIDLib.h`) to the I/O Kit function `IOServiceMatching`. This defines a broad search for all objects in the I/O Registry that represent HID class devices. Then it calls the I/O Kit function `IOServiceGetMatchingServices`, which performs the search and, if successful, returns an iterator for the set of matching devices.

The parameters to `MyFindHIDDevices` are a Mach port and a pointer to an object iterator. The Mach port is obtained in a previous call to the `IOMasterPort` function—see Listing 3-2 (page 55). The `MyFindHIDDevices` function uses the pointer to an object iterator to return an iterator that provides access to matching driver objects from the I/O Registry.

Listing 3-3 A function that searches for HID devices that match the specified criteria

```
static void MyFindHIDDevices(mach_port_t masterPort,
                io_iterator_t *hidObjectIterator)
{
    CFMutableDictionaryRef hidMatchDictionary = NULL;
    IOReturn ioReturnValue = kIOReturnSuccess;
    Boolean noMatchingDevices = false;

    // Set up a matching dictionary to search the I/O Registry by class
    // name for all HID class devices
    hidMatchDictionary = IOServiceMatching(kIOHIDDeviceKey);

    // Now search I/O Registry for matching devices.
    ioReturnValue = IOServiceGetMatchingServices(masterPort,
                            hidMatchDictionary, hidObjectIterator);

    noMatchingDevices = ((ioReturnValue != kIOReturnSuccess)
                                | (*hidObjectIterator == NULL));

    //If search is unsuccessful, print message and hang.
    if (noMatchingDevices)
        print_errmsg_if_err(ioReturnValue, "No matching HID class devices
found.");

    // IOServiceGetMatchingServices consumes a reference to the
    //   dictionary, so we don't need to release the dictionary ref.
    hidMatchDictionary = NULL;
}
```

The `MyFindHIDDevices` function doesn't release the reference it obtains to a matching dictionary because when it calls the I/O Kit function `IOServiceGetMatchingServices`, that function consumes a reference to the dictionary.

Working With Legacy HID Class Device Interfaces

This chapter provides step-by-step instructions, including listings of sample code, for accessing a HID class device on older versions of Mac OS X. If you are developing new code for Mac OS X v10.5 or later, you should use the APIs described in "Accessing a HID Device" (page 13)..

These examples assume that you have already obtained an iterator for a list of devices matching certain properties (for example, by using the `MyFindHIDDevices` function, Listing 3-3 (page 56)).

Now that you have a device iterator, you can use it to examine each device and create a device interface for it. The functions in this chapter demonstrate this process by first displaying the properties and then creating and testing a device interface for each device in turn.

This chapter contains the following sections:

For code to help you locate HID class devices, see "Legacy HID Access Overview" (page 53).

Printing the Properties of a HID Class Device

Since the properties of a HID class device often contain nested element dictionaries, the functions that access the elements must do so in a recursive manner. In the sample code, the `MyShowHIDProperties` function uses CFShow to print a dictionary.

Listing 4-1 A function that shows all properties for a HID class device

```
static void MyShowHIDProperties(io_registry_entry_t hidDevice)
{
        kern_return_t                                   result;
        CFMutableDictionaryRef                          properties = 0;
        char                                            path[512];

        result = IORegistryEntryGetPath(hidDevice, kIOServicePlane, path);
        if (result == KERN_SUCCESS)
                printf("[ %s ]", path);

        //Create a CF dictionary representation of the I/O
        //Registry entry's properties
```

```
        result = IORegistryEntryCreateCFProperties(hidDevice, &properties,
                kCFAllocatorDefault, kNilOptions);
        if ((result == KERN_SUCCESS) && properties)
        {
                CFShow(properties);
                /* Some common properties of interest include:
                    kIOHIDTransportKey, kIOHIDVendorIDKey,
                    kIOHIDProductIDKey, kIOHIDVersionNumberKey,
                    kIOHIDManufacturerKey, kIOHIDProductKey,
                    kIOHIDSerialNumberKey, kIOHIDLocationIDKey,
                    kIOHIDPrimaryUsageKey, kIOHIDPrimaryUsagePageKey,
                    and kIOHIDElementKey.
                */

                //Release the properties dictionary
                CFRelease(properties);
        }
        printf("\n\n");
}
```

`MyShowHIDProperties` takes the HID class device found in the I/O Registry and uses `IORegistryEntryCreateCFProperties` to create a CF dictionary representation of the device's properties. It then calls CFShow to format the information and print it to the screen.

Creating a HID Class Device Interface

A device interface provides functions your application or other code running on Mac OS X can use to access a device. The `MyCreateHIDDeviceInterface` function, shown in Listing 4-2 (page 59), creates and returns a HID class device interface based on a passed object from the I/O Registry. It also demonstrates how to obtain the class name of the passed object by calling the I/O Kit function `IOObjectGetClass`.

To create a HID class device interface, the `MyCreateHIDDeviceInterface` function performs the following steps:

1. It obtains an interface of type `IOCFPlugInInterface` for the HID class device represented by the passed `hidDevice` object.

   To obtain this interface, it calls the `IOCreatePlugInInterfaceForService` function, passing the value `kIOHIDDeviceUserClientTypeID` for the plug-in type parameter and the value `kIOCFPlugInInterfaceID` for the interface type parameter. `kIOHIDDeviceUserClientTypeID` is defined in `IOHIDLib.h` and `kIOCFPlugInInterfaceID` is defined in `IOCFPlugIn.h` (located in `IOKit.framework`).

2. It obtains a HID class device interface by calling the `QueryInterface` method of the `IOCFPlugInInterface` object. To specify a HID class device interface, it uses the following term:

   ```
   CFUUIDGetUUIDBytes(kIOHIDDeviceInterfaceID)
   ```

   This term produces a UUID (universally unique identifier) for the device interface. UUIDs are described in the Core Foundation developer documentation, available in the Reference Library > Core Foundation section of the developer documentation website. `kIOHIDDeviceInterfaceID` is defined in `IOHIDLib.h`.

3. It releases the `IOCFPlugInInterface` object and returns the HID class device interface in the `hidDeviceInterface` parameter.

Note: The following listing uses a number of minor support functions that are not included as part of the text of this document because they do not relate directly to the use of HID class device interfaces. For a complete listing that includes these support functions, see "Complete Code Samples" (page 69).

Listing 4-2 A function that creates and returns a HID class device interface based on a passed object from the I/O Registry

```
static void MyCreateHIDDeviceInterface(io_object_t hidDevice,
                            IOHIDDeviceInterface ***hidDeviceInterface)
{
    io_name_t               className;
    IOCFPlugInInterface     **plugInInterface = NULL;
    HRESULT                 plugInResult = S_OK;
    SInt32                  score = 0;
    IOReturn                ioReturnValue = kIOReturnSuccess;

    ioReturnValue = IOObjectGetClass(hidDevice, className);

    print_errmsg_if_io_err(ioReturnValue, "Failed to get class name.");

    printf("Found device type %s\n", className);

    ioReturnValue = IOCreatePlugInInterfaceForService(hidDevice,
                            kIOHIDDeviceUserClientTypeID,
                            kIOCFPlugInInterfaceID,
                            &plugInInterface,
                            &score);

    if (ioReturnValue == kIOReturnSuccess)
    {
        //Call a method of the intermediate plug-in to create the device
        //interface
        plugInResult = (*plugInInterface)->QueryInterface(plugInInterface,
                            CFUUIDGetUUIDBytes(kIOHIDDeviceInterfaceID),
                            (LPVOID *) hidDeviceInterface);
        print_errmsg_if_err(plugInResult != S_OK, "Couldn't create HID class
device interface");

        (*plugInInterface)->Release(plugInInterface);
    }
}
```

Obtaining HID Cookies

The following function, `getHIDCookies` (shown in Listing 4-3 (page 60)), places certain cookies associated with the HID class device's x axis, button 1, button 2, and button 3 into a data structure so the queue handling functions can add these elements to a queue.

Listing 4-3 A function that stores selected cookies in global variables for later use

```
typedef struct cookie_struct
{
        IOHIDElementCookie gXAxisCookie;
        IOHIDElementCookie gButton1Cookie;
        IOHIDElementCookie gButton2Cookie;
        IOHIDElementCookie gButton3Cookie;
} *cookie_struct_t;

cookie_struct_t getHIDCookies(IOHIDDeviceInterface122 **handle)
{
        cookie_struct_t cookies = memset(malloc(sizeof(*cookies)), 0,
            sizeof(*cookies));
        CFTypeRef                               object;
        long                                    number;
        IOHIDElementCookie                      cookie;
        long                                    usage;
        long                                    usagePage;
        CFArrayRef                              elements; //
        CFDictionaryRef                         element;
        IOReturn                                success;

        if (!handle || !(*handle)) return cookies;

        // Copy all elements, since we're grabbing most of the elements
        // for this device anyway, and thus, it's faster to iterate them
        // ourselves. When grabbing only one or two elements, a matching
        // dictionary should be passed in here instead of NULL.
        success = (*handle)->copyMatchingElements(handle, NULL, &elements);

        printf("LOOKING FOR ELEMENTS.\n");
        if (success == kIOReturnSuccess) {
                CFIndex i;
                printf("ITERATING...\n");
                for (i=0; i<CFArrayGetCount(elements); i++)
                {
                        element = CFArrayGetValueAtIndex(elements, i);
                        // printf("GOT ELEMENT.\n");

                        //Get cookie
                        object = (CFDictionaryGetValue(element,
                            CFSTR(kIOHIDElementCookieKey)));
                        if (object == 0 || CFGetTypeID(object) != CFNumberGetTypeID())
                            continue;
                        if(!CFNumberGetValue((CFNumberRef) object, kCFNumberLongType,
                            &number))
                                continue;
                        cookie = (IOHIDElementCookie) number;

                        //Get usage
                        object = CFDictionaryGetValue(element,
CFSTR(kIOHIDElementUsageKey));
                        if (object == 0 || CFGetTypeID(object) != CFNumberGetTypeID())
                            continue;
                        if (!CFNumberGetValue((CFNumberRef) object, kCFNumberLongType,
                            &number))
                                continue;
```

```
                usage = number;

                //Get usage page
                object = CFDictionaryGetValue(element,
                    CFSTR(kIOHIDElementUsagePageKey));
                if (object == 0 || CFGetTypeID(object) != CFNumberGetTypeID())
                    continue;
                if (!CFNumberGetValue((CFNumberRef) object, kCFNumberLongType,
                    &number))
                        continue;
                usagePage = number;

                //Check for x axis
                if (usage == 0x30 && usagePage == 0x01)
                    cookies->gXAxisCookie = cookie;
                //Check for buttons
                else if (usage == 0x01 && usagePage == 0x09)
                    cookies->gButton1Cookie = cookie;
                else if (usage == 0x02 && usagePage == 0x09)
                    cookies->gButton2Cookie = cookie;
                else if (usage == 0x03 && usagePage == 0x09)
                    cookies->gButton3Cookie = cookie;
            }
            printf("DONE.\n");
        } else {
            printf("copyMatchingElements failed with error %d\n", success);
        }

        return cookies;
}
```

Performing Operations on a HID Class Device

Once you have found a HID class device in the I/O Registry and created a device interface for it, you can perform operations such as:

- opening the device
- getting element values
- creating and manipulating queues
- closing the device

The sample code in this section performs these types of operations through the test function `MyTestHIDDeviceInterface`, shown in Listing 4-4 (page 61). This function is passed a HID class device interface and it first calls the device interface function `open` to open the device. Next, it calls `MyTestQueues`, shown in Listing 4-5 (page 62), to create and manipulate a queue. Then, it calls `MyTestHIDInterface` to get various element values and print them. Finally, it closes the device by calling the device interface function `close`.

Listing 4-4 A function that performs testing on a passed HID class device interface

```
static void MyTestHIDDeviceInterface(IOHIDDeviceInterface **hidDeviceInterface,
cookie_struct_t cookies)
```

```
{
        IOReturn ioReturnValue = kIOReturnSuccess;

        //open the device
        ioReturnValue = (*hidDeviceInterface)->open(hidDeviceInterface, 0);
        printf("Open result = %d\n", ioReturnValue);

        //test queue interface
        MyTestQueues(hidDeviceInterface, cookies);

        //test the interface
        MyTestHIDInterface(hidDeviceInterface, cookies);

        //close the device
        if (ioReturnValue == KERN_SUCCESS)
                ioReturnValue = (*hidDeviceInterface)->close(hidDeviceInterface);

        //release the interface
        (*hidDeviceInterface)->Release(hidDeviceInterface);
}
```

The `MyTestQueues` function (shown in Listing 4-5 (page 62)) first calls the HID device interface function `allocQueue` to allocate a queue. Next, it uses the queue handling functions `addElement`, `hasElement`, and `removeElement` to add elements to the queue, check to see if an element is in the queue, and remove an element, respectively. Finally, `MyTestQueues` simulates a game loop by calling the `start` queue function to start data delivery to the queue and then repeatedly calling the `getNextEvent` queue function to retrieve the values for elements in the queue. All queue handling functions for HID class devices are defined in `IOHIDLib.h`.

Listing 4-5 A function to test the HID class device interface queue handling functions

```
static void MyTestQueues(IOHIDDeviceInterface **hidDeviceInterface, cookie_struct_t
cookies)
{
        HRESULT                                                 result;
        IOHIDQueueInterface **                                          queue;
        Boolean                                         hasElement;
        long                                            index;
        IOHIDEventStruct                                                event;

        queue = (*hidDeviceInterface)->allocQueue(hidDeviceInterface);

        if (queue)
        {
                printf("Queue allocated: %lx\n", (long) queue);
                //create the queue
                result = (*queue)->create(queue, 0, 8);
                    /*  depth (8): maximum number of elements in queue before oldest
                        elements in queue begin to be lost.
                     */
                printf("Queue create result: %lx\n", result);

                //add elements to the queue
                result = (*queue)->addElement(queue, cookies->gXAxisCookie, 0);
                printf("Queue added x axis result: %lx\n", result);
                result = (*queue)->addElement(queue, cookies->gButton1Cookie, 0);
                printf("Queue added button 1 result: %lx\n", result);
```

```
        result = (*queue)->addElement(queue, cookies->gButton2Cookie, 0);
        printf("Queue added button 2 result: %lx\n", result);
        result = (*queue)->addElement(queue, cookies->gButton3Cookie, 0);
        printf("Queue added button 3 result: %lx\n", result);


        //check to see if button 3 is in queue
        hasElement = (*queue)->hasElement(queue,cookies->gButton3Cookie);
        printf("Queue has button 3 result: %s\n", hasElement ? "true" :
            "false");

        //remove button 3 from queue
        result = (*queue)->removeElement(queue, cookies->gButton3Cookie);
        printf("Queue remove button 3 result: %lx\n",result);

        //start data delivery to queue
        result = (*queue)->start(queue);
        printf("Queue start result: %lx\n", result);

        //check queue a few times to see accumulated events
        sleep(1);
        printf("Checking queue\n");
        for (index = 0; index < 10; index++)
        {
                AbsoluteTime                            zeroTime = {0,0};

                result = (*queue)->getNextEvent(queue, &event, zeroTime, 0);
                if (result)
                        printf("Queue getNextEvent result: %lx\n", result);
                else
                        printf("Queue: event:[%lx] %ld\n",
                                                        (unsigned long)
event.elementCookie,
                                                        event.value);

                sleep(1);
        }

        //stop data delivery to queue
        result = (*queue)->stop(queue);
        printf("Queue stop result: %lx\n", result);

        //dispose of queue
        result = (*queue)->dispose(queue);
        printf("Queue dispose result: %lx\n", result);

        //release the queue we allocated
        (*queue)->Release(queue);
    }
}
```

The `MyTestHIDInterface` function uses the passed-in HID device interface to call the device interface `getElement` function to get the element values associated with the cookies stored in the global variables `gXAxisCookie`, `gButton1Cookie`, `gButton2Cookie`, and `gButton3Cookie` by `getHIDCookies` (shown in Listing 4-3 (page 60)). The sample code simulates a game loop by repeatedly calling `getElementValue` in a `for` loop.

Note: HID device interfaces have two functions, `getElementValue` and `queryElementvalue`. They behave in subtly different ways, and should not be confused, for performance reasons.

The function `getElementValue` should be used for elements associated with inputs (button presses, for example). These values are typically sent whenever the value changes via interrupt reports. Thus, the HID stack will return the last value sent by the device.

The function `queryElementvalue` should be used only for "feature" elements. Since these elements do not trigger an interrupt, their values must be manually polled from the device.

Listing 4-6 A function that uses HID class device interface functions to get and display selected element values over time

```
static void MyTestHIDInterface(IOHIDDeviceInterface ** hidDeviceInterface, cookie_struct_t
 cookies)
{
        HRESULT                                 result;
        IOHIDEventStruct                                        hidEvent;
        long                                    index;

        printf("X Axis (%lx), Button 1 (%lx), Button 2 (%lx), Button 3 ($lx)\n",
                        (long) cookies->gXAxisCookie, (long) cookies->gButton1Cookie,
                       (long) cookies->gButton2Cookie, (long) cookies->gButton3Cookie);

        for (index = 0; index < 10; index++)
        {
                long xAxis, button1, button2, button3;

                //Get x axis
                result = (*hidDeviceInterface)->getElementValue(hidDeviceInterface,
                    cookies->gXAxisCookie, &hidEvent);
                if (result)
                     printf("getElementValue error = %lx", result);
                xAxis = hidEvent.value;

                //Get button 1
                result = (*hidDeviceInterface)->getElementValue(hidDeviceInterface,
                    cookies->gButton1Cookie, &hidEvent);
                if (result)
                     printf("getElementValue error = %lx", result);
                button1 = hidEvent.value;

                //Get button 2
                result = (*hidDeviceInterface)->getElementValue(hidDeviceInterface,
                    cookies->gButton2Cookie, &hidEvent);
                if (result)
                     printf("getElementValue error = %lx", result);
                button2 = hidEvent.value;

                //Get button 3
                result = (*hidDeviceInterface)->getElementValue(hidDeviceInterface,
                    cookies->gButton3Cookie, &hidEvent);
                if (result)
                     printf("getElementValue error = %lx", result);
                button3 = hidEvent.value;

                //Print values
```

```
            printf("%ld %s%s%s\n", xAxis, button1 ? "button1 " : "",
                           button2 ? "button2 " : "", button3 ? "button3 " : "");

            sleep(1);
        }
}
```

Using Queue Callbacks

There are two basic ways of using queues when interacting with HID devices: polling and callbacks. In Listing 4-5 (page 62), queues are accessed in a polled fashion. The example project "hidexample" uses this mechanism.

For performance reasons (and for programming convenience), it often makes more sense to use queue callbacks. In this usage, a function in your program is called whenever the queue becomes non-empty.

The code snippet is relatively straightforward. Essentially, you allocate a private data structure that contains a reference to the queue, create an asynchronous event source for the queue, add a queue callback handler, and finally, add the newly-created queue event source to your run loop.

The queue callback is passed two `void *` parameters, `callbackRefcon` and `callbackTarget`, both of which can contain pointers to arbitrary data. With this design, the same callback function can operate on multiple queues, feeding data to multiple class instances, simply by passing different queue and class pointers to the `setEventCallout` function.

In Listing 4-7 (page 65), the `callbackTarget` parameter will be passed `NULL`, and the `callbackRefcon` parameter will be passed a dynamically-allocated object containining the `hidQueueInterface` pointer (just for grins).

Listing 4-7 Adding a Queue Callback

```
bool addQueueCallbacks(IOHIDQueueInterface **hidQueueInterface)
{
    IOReturn ret;
    CFRunLoopSourceRef eventSource;
    /*  We could use any data structure here. This data structure
        will be passed to the callback, and should probably
        include some information about the queue, assuming your
        program deals with more than one. */
    IOHIDQueueInterface ***myPrivateData = malloc(sizeof(*myPrivateData));
    *myPrivateData = hidQueueInterface;

    // In the calling function, we did something like:
    // hidQueueInterface = (*hidDeviceInterface)->
    //      allocQueue(hidDeviceInterface);
    // (*hidQueueInterface)->create(hidQueueInterface, 0, 8);
    ret = (*hidQueueInterface)->
        createAsyncEventSource(hidQueueInterface,
        &eventSource);
    if ( ret != kIOReturnSuccess )
        return false;
    ret = (*hidQueueInterface)->
        setEventCallout(hidQueueInterface,
        QueueCallbackFunction, NULL, myPrivateData);
    if ( ret != kIOReturnSuccess )
```

```
            return false;
        CFRunLoopAddSource(CFRunLoopGetCurrent(), eventSource,
            kCFRunLoopDefaultMode);
        return true;
    }
```

The final example in this section is taken from the "HidTestTool" example. This snippet shows how to handle a queue callback. This code is structurally somewhat different in that a data structure is passed as the `refcon` argument. This allows it to pass in both information about the queue and information about the elements supported by a given HID device.

Listing 4-8 A Basic Queue Callback Function

```
static void QueueCallbackFunction(
                            void *          target,
                            IOReturn            result,
                            void *          refcon,
                            void *          sender)
{
    HIDDataRef          hidDataRef      = (HIDDataRef)refcon;
    AbsoluteTime    zeroTime    = {0,0};
    CFNumberRef     number      = NULL;
    CFMutableDataRef    element     = NULL;
    HIDElementRef   tempHIDElement  = NULL;//(HIDElementRef)refcon;
    IOHIDEventStruct    event;
    bool                change;
    bool                stateChange = false;

    if ( !hidDataRef || ( sender != hidDataRef->hidQueueInterface))
        return;

    while (result == kIOReturnSuccess)
    {
        result = (*hidDataRef->hidQueueInterface)->getNextEvent(
                                    hidDataRef->hidQueueInterface,
                                    &event,
                                    zeroTime,
                                    0);

        if ( result != kIOReturnSuccess )
            continue;

        // Only intersted in 32 values right now
        if ((event.longValueSize != 0) && (event.longValue != NULL))
        {
            free(event.longValue);
            continue;
        }

        number = CFNumberCreate(kCFAllocatorDefault, kCFNumberIntType,
            &event.elementCookie);
        if ( !number )  continue;
        element = (CFMutableDataRef)CFDictionaryGetValue(hidDataRef->
            hidElementDictionary, number);
        CFRelease(number);

        if ( !element ||
            !(tempHIDElement = (HIDElement *)CFDataGetMutableBytePtr(element)))
```

```
            continue;

        change = (tempHIDElement->currentValue != event.value);
        tempHIDElement->currentValue = event.value;

    }

}
```

Complete Code Samples

The code samples elsewhere in this document are intended to introduce concepts. Thus, this document does not include all the necessary code to build a working tool on its own.

The code samples in this chapter are complete samples that you can run and use as a basis for experimentation.

Current Examples

The following sample code projects show how to use the HID Manager interfaces added in Mac OS X v10.5:

- *HID Explorer*—A Cocoa application that demonstrates how to use the HID Manager APIs while providing a useful tool for exploring the HID devices attached to your computer.
- *HID Config Save*—A Carbon application that demonstrates how to save and restore element configuration information and shows how to use the configured inputs in a game environment.
- *HID Calibrator*—A Carbon application that demonstrates the HID Manager APIs..
- *HID LED test tool*—A command-line tool that demonstrates how to iterate HID devices and manipulate their LEDs.

Legacy Examples

The code samples in this section are designed for older versions of Mac OS X (prior to version 10.5). For version 10.5 and later, these APIs are deprecated. You should use the APIs shown in "Accessing a HID Device" (page 13) for new code written for Mac OS X v10.5 and later.

The Companion Files disk image contains a complete, buildable version of the example in "Legacy HID Access Overview" (page 53) and "Working With Legacy HID Class Device Interfaces" (page 57).

In addition, a second example tool, HIDTestTool, is included. This tool provides additional sample code that may be useful when working with various HID devices.

If you are viewing this document on the web, the disk image can be downloaded by clicking the "Companion Files" link at the top of the table of contents. If you are viewing this document as a PDF file, or if you are viewing it from a local installation, you must first go to the online version of this document at http://developer.apple.com/documentation/DeviceDrivers/Conceptual/HID/index.html.

In addition to these examples, the *HID Manager Basics*, *HID Utilities Source*, and *UniversalHIDModuleTest* sample code projects use these legacy APIs.

Document Revision History

This table describes the changes to *HID Class Device Interface Guide*.

Date	Notes
2009-10-19	Updated to cover the new HID Manager APIs in Mac OS X v10.5.
2009-05-06	Made minor corrections to code samples.
2006-10-03	Clarified the location of sample files.
2006-03-08	Changed section titling to better reflect contents.
2005-11-09	Corrected the name of an example function; removed smart quotes in code listings.
2005-08-11	Made minor typographical fixes.
2005-07-07	Made minor wording changes for clarity.
2005-06-04	Added queue callback examples. Improved overall structure.
2005-04-29	Updated content to reflect more modern APIs and coding practices recommended for Mac OS X 10.3 and later.
2001-05-01	Fixed broken links
	Earliest known revision. Release history prior to 2001 is unavailable.

Document Revision History