HID Class Device Interface Guide

Hardware & Drivers > Human Interface Device & Force Feedback

2006-10-03

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 HID Class Device Interfaces" (page 17) 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 `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`.

Many 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. All 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 can 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.

HID Access Overview

This chapter provides step-by-step instructions, including listings of sample code, for accessing a HID class device on Mac OS X. 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 13)
2. "Setting Up Your Programming Environment" (page 14)
3. "Accessing the I/O Kit Master Port" (page 15)
4. "Finding a HID Class Device" (page 15)

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" when creating your project. Among other things, this causes the project to include `CoreFoundation.framework` under External Frameworks and Libraries.
2. Use Add Frameworks... from the Projects menu to add `IOKit.framework` and `System.framework` to your project.
3. It's 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 29).

Listing 2-1 (page 14) 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 2-1 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 2-2 (page 15). 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 2-3 (page 16), 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 2-2 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 to 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 2-3 (page 16)) 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 2-2 (page 15). 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 2-3 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 HID Class Device Interfaces

This chapter provides step-by-step instructions, including listings of sample code, for accessing a HID class device on Mac OS X. 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 2-3 (page 16)).

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:

1. "Printing the Properties of a HID Class Device" (page 17)
2. "Creating a HID Class Device Interface" (page 18)
3. "Obtaining HID Cookies" (page 20)
4. "Performing Operations on a HID Class Device" (page 21)
5. "Using Queue Callbacks" (page 25)

For code to help you locate HID class devices, see "HID Access Overview" (page 13).

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 3-1 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 3-2 (page 19), 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 29).

Listing 3-2 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 3-3 (page 20)), 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 3-3 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 3-4 (page 22). 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 3-5 (page 22), 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 3-4 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 3-5 (page 22)) 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 3-5 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 3-3 (page 20)). 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 3-6 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 3-5 (page 22), 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 3-7 (page 25), 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 3-7 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 3-8 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 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 Companion Files disk image contains a complete, buildable version of this code example.

In addition, a second example tool, HIDTestTool, is included, which 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.

Document Revision History

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

Date	Notes
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