Mac Custom HID Demo

Contents

Xcode and Objective-C

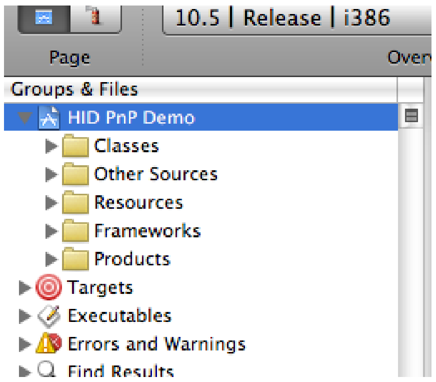
The Mac Custom HID Demo application was created using Xcode, Apple's IDE that comes pre-packaged with Mac OS X operating systems. In this example, the Cocoa application programming interface (API) is used. Cocoa is one of Apple's APIs available for Mac OS X, and uses the Objective-C programming language. Objective-C is an object-oriented programming language that is built on top of C that is used primarily by Apple. For more information on Objective-C and a good starting point for beginners, check out Apple's guide to Learning Objective-C.

Interface Builder

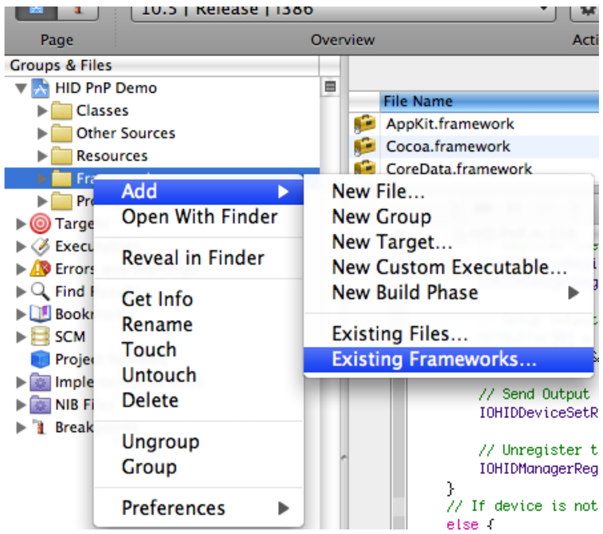
The Interface Builder application is part of Xcode, and also comes pre-packaged with MAC OS X operating systems. It is used to design graphical user interfaces (GUIs), and was used to design the GUI in this example. A simple step-by-step tutorial showing how to use Xcode with the Interface Builder is available in the Mac OS X Reference Library called the Cocoa Application Tutorial.

HID Framework

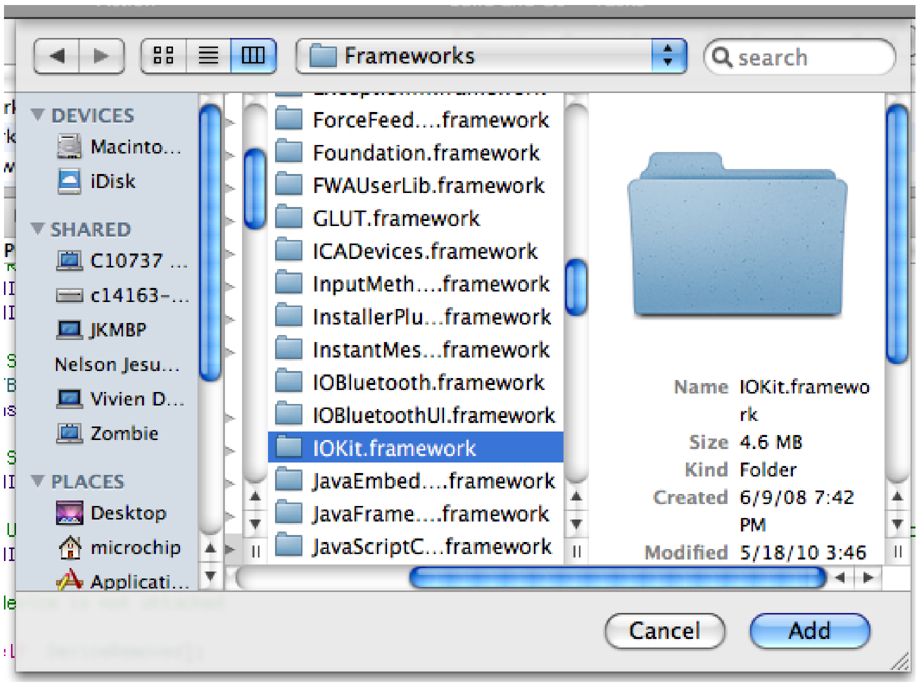
For HID class USB communication between a Mac and a connected device, Cocoa uses an API called the HID Manager that is part of the I/O Kit framework. In order to use the HID Manager in your project, the I/O Kit framework needs to be added. Follow these steps to do this:



1. After creating your project, expand your project in the “Groups & Files” panel to show the Frameworks subfolder.

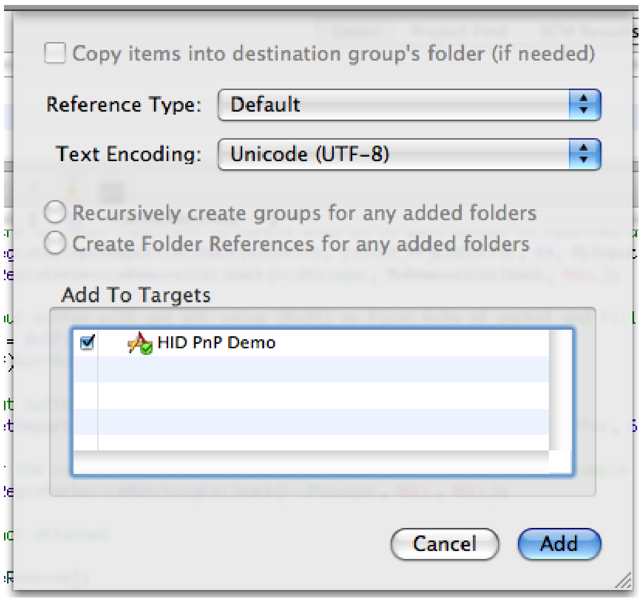


2. Right click on the Frameworks subfolder (command-click) and select Add > Existing Frameworks…



Name IOKit.framework
Size 4.6 MB
Kind Folder
Created 6/9/08 7:42 PM
Modified 5/18/10 3:46

3. Scroll down and select IOKit.framework and click the Add button.



Copy items into destination group's folder (if needed)
Reference Type: Default
Text Encoding: Unicode (UTF-8)
Recursively create groups for any added folders
Create Folder References for any added folders
Add To Targets
✓ HID PnP Demo

4. A verification window will be shown. Click the Add button again.

Now the HID Manager can be used in your project. To make the functions within the HID Manager API available make sure to use #import. #import is similar to #include, but ensures a header is included only once in a source file. For this example, the HID_PnP class uses the following:

```
#import <IOKit/hid/IOHIDManager.h>
```

This API is implemented in C, but because Objective-C is a strict superset of C, the API can be used interchangeably within Objective-C code. The HID Manager handles all enumerated HID class devices and allows access to their properties, notification for when a new device is enumerated or an existing devices is removed, and communication between host and device. For more detailed information about the HID Manager, both Technical Note TN2187 and the HID Class Device Interface Guide are good sources.

Application Overview

Overall, the same GUI and functionality is used as Microchip's Plug and Play example for Windows. The pushbutton status and potentiometer values are transmitted from the device to the host, and when the toggle LED(s) button is pressed in the app window, the LEDs on the device are toggled. The demo also supports hot plugging the device and detects when the device is removed.

Because the demo is written in a different language and is on a different operating system, the code layout is not exactly the same. This is a brief overview of how the application works. There are two objective-C classes that behave similarly to a C++ class or Java class, the HID_PnP class and the DemoApp class.

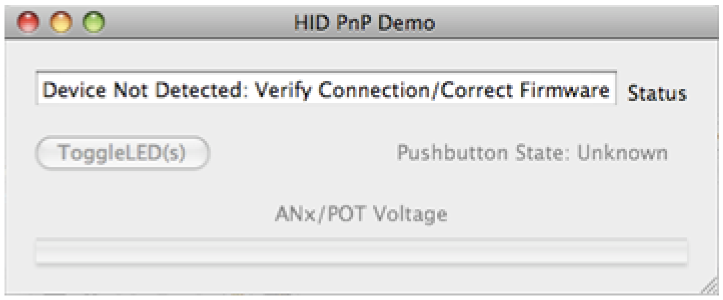
The HID_PnP class handles the communication with the Custom HID Device. There is an instance of a HID Manager as well as an instance of a HID Device within the class. When a device is attached, the HID Device instance holds the information about that device. The HID Manager is always running and is initialized with a few C callback functions for when a device is attached, removed, or when it sends data. These functions are called when one of these events occurs. Otherwise it returns to the run loop that is running the application.

The DemoApp class handles the application window created in the interface builder. It updates what is seen in the window depending on what information is sent from the device. There is an object of the HID_PnP class that is part of the DemoApp class. This object handles the communication and is polled every 15 ms (~60 Hz) by the DemoApp Class to update its window. It also notifies the HID_PnP object when the Toggle LED(s) button has been pressed.

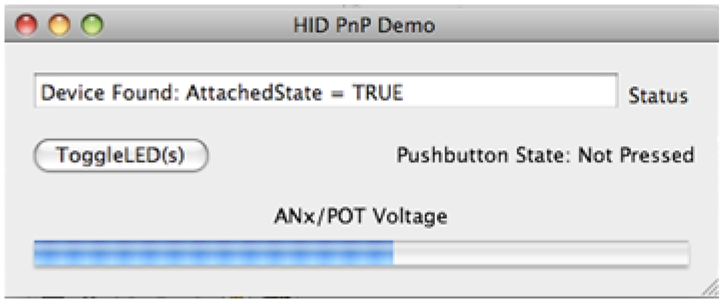
The run loop for the entire application is shared by HID communication, GUI updates, and the button in the GUI window. This means that all of these are in one thread, which is different from the windows application. The windows application has two threads, one for updating the GUI and waiting for a button press, and another for HID communication. The windows thread that handles the HID communication communicates asynchronously with the device. This means that the functions called are blocking, so while the host is communicating the device, any other code is halted from executing on the same thread. For the mac, the communication is synchronous, which means that the functions are non-blocking and perform similar to an interrupt on a microcontroller. This means that the application can return to the run loop and perform other tasks such as updating the gui while it waits for the device to respond to the host.

Graphical User Interface Overview

Below is what the application window looks like when the device is disconnected. The progress bar and labels are greyed out, the toggle LED button is disabled, the text box at the top states that the device is not attached, and the pushbutton status is unknown.


HID PnP Demo
Device Not Detected: Verify Connection/Correct Firmware
Status
ToggleLED(s)
Pushbutton State: Unknown
ANx/POT Voltage

Next is what the application window looks like when the device is connected. The progress bar begins to show the value of the potentiometer, the labels are no longer greyed out, the toggle LED button is no longer disabled, the text box at the top states that the device is attached, and the pushbutton state is either pressed or unpressed.


HID PnP Demo
Device Found: AttachedState = TRUE
Status
ToggleLED(s)
Pushbutton State: Not Pressed
ANx/POT Voltage

Common Pitfalls

Mixing Objective-C with C

There are some common pitfalls that one might run into while developing USB code for the mac. For instance, if an application GUI is preferred, then the interface builder is the easiest way to create the GUI. In order to use the interface builder, objective-C should be used. However, some APIs are in C or C++, which means that mixing the two languages is required. It is possible to have C functions inside of an objective C class.

This example shows how that is possible. The C callback functions for the HID Manager are part of the HID_PnP class. However, since they are C functions, they do not have access to the variables inside of the class nor can they have objects including the instance of the HID_PnP class which is referred to as "self" inside of the class. Because "self" can't be used, accessing the class methods is difficult (which is called sending a message in Objective-C). To go around this, a self referencing void pointer is created as a global variable in the source file of the HID_PnP class. This pointer points to the instance of the class itself in the init method – the class initialization function for HID_PnP. Because of Objective-C's ability for dynamic typing (see Objective-C overview), the void pointer can act as the self reference and can access the class's methods.

Additionally, because some functions are in C, they are unable to take objects as arguments. To go around this, there are definitions of structs within the API that are the same size as objects that are needed as arguments. So, if writing in objective-C type-casting can be used to go between objects and structs.

Mouse Click Events

For each thread used in an application there is a run loop. A run loop is (usually) an infinite loop that runs certain actions over and over again. However, if one of these actions is blocking (meaning it waits for an event to complete before returning to the run loop) then it can create a delay for other actions within the same run loop. The usual solution to this problem is to separate the blocking actions into separate threads. This is not always the case, though.

A mouse click event is when a mouse click occurs in the window that can be blocking to the application, for example clicking and holding down a button, or dragging or resizing the window. These events can disrupt the workflow of the application. Instead of creating an additional thread so that all other functions can run on an additional run loop, the functions can be within the same run loop but with a certain run mode. When you add an action to a run loop, you add it with a mode. Typically the default mode is used (kCFRunLoopDefaultMode). However, if you add the action to the run loop with the event tracking mode (NSEventTrackingRunLoopMode), then it will still be able to run while a mouse click event is occurring. Therefore, an action needs to be added to the run loop twice, once with the default mode, and second with the event tracking mode. For this example there are three actions and one run loop. The mouse click event, which is when the button is being pressed or the window is being resized, the timer that updates the GUI with the HID communication from the device, and the HID Manager which handles the HID communication. Both the timer and the HID Manager were added to the run loop in both modes, so they won't be disrupted by a mouse click event.

CPU Usage

As stated in the previous section, a run loop contains actions that are continuously repeated. If these actions utilize the CPU fully, then your application can end up hogging most of the CPU resources. If the actions only need to be performed periodically, then a great way to free up CPU resources is to suspend the thread's execution for a period of time. This can be done with any of the three sleep subroutines: sleep or nsleep. The sleep function has the following form:

unsigned int sleep(unsigned int seconds);

The thread is suspended for the specified amount of seconds or until a signal is received. It returns a zero if the requested time has elapsed, or the number of seconds that are left if a signal is received. The nsleep function takes in two timespec structs and has the following form:

*int nsleep(const struct timespec *req, struct timespec *rem);*

And the timespec struct has the following form:

```
struct timespec {
        time_t  tv_sec;         /* seconds */
        long    tv_nsec;        /* nanoseconds */
};
```

The nsleep function sleeps for the specified amount of time given in *req or until a signal is received. It returns 0 after completing the amount of time or -1 if a signal has been received. If a signal has been received, the remaining time is written to the struct that *rem is pointing to.