Building a HID Motion Sensor

with the Netduino Plus

August 16, 2013

Abstract

This paper describes how to build a simple HID device that you can attach to your Windows 8.1 tablet or laptop. It provides guidelines for configuring the development board, building the circuit, and creating the firmware. It assumes that the reader is familiar with the HID protocol, programmable microcontrollers, and the .Net Micro Framework.

This information applies to the following operating systems:   
 Windows 8.1

References and resources discussed here are listed at the end of this paper.

© 2013 Microsoft Corporation. All rights reserved.

**Disclaimer**: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet website references, may change without notice. Some information relates to pre-released product which may be substantially modified before it’s commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here. You bear the risk of using it. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

The starting point for the firmware was samples found on the Netduino forum:

<http://forums.netduino.com/index.php?/topic/1514-driverless-pc-netduino-communication-using-usb/>

Copyright (c) Secret Labs LLC. All Rights Reserved.

Licensed under the Apache 2.0 open source license

<http://forums.netduino.com/index.php?/topic/1246-parallax-pir-sensor-class/>

Licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License



Contents

[Introduction 3](#_Toc363044985)

[Quickstart 3](#_Toc363044986)

[Hardware Parts List 3](#_Toc363044987)

[Preparing your Netduino Plus 5](#_Toc363044988)

[Building the circuit 5](#_Toc363044989)

[Attaching the RS232 Shield 5](#_Toc363044990)

[Attaching the Passive Infrared sensor 5](#_Toc363044991)

[Installing the device firmware 6](#_Toc363044992)

[Identifying the COM port 6](#_Toc363044993)

[Completing the installation 7](#_Toc363044994)

[The device firmware 10](#_Toc363044995)

[Supporting the USB transport 11](#_Toc363044996)

[Defining the Sensor class 11](#_Toc363044997)

[Opening the device connection 12](#_Toc363044998)

[Closing the device connection 12](#_Toc363044999)

[Supporting the HID protocol 12](#_Toc363045000)

[Supporting the HID input report 13](#_Toc363045001)

[Supporting the HID output report 13](#_Toc363045002)

[Creating the Report Descriptor 14](#_Toc363045003)

[Integrating the motion sensor 15](#_Toc363045004)

[Sensor events 16](#_Toc363045005)

[The report interval 17](#_Toc363045006)

[Resources 17](#_Toc363045007)

# Introduction

With the addition of the new [HID WinRT API](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.aspx) for Windows 8.1, Microsoft has given academics, hobbyists, and industrial engineers a great platform for connecting their remote devices to tablets and laptops. The new API simplifies the integration of a broad spectrum of devices: from measurement equipment in the scientific lab, to weather stations, to medical devices, to exercise devices.

The one common thread is that these devices all support the HID protocol. In its earliest days, the protocol supported: keyboards, mice, and joysticks—devices that acted as a “human interface” to a computer. Hence the name “Human Interface Device”, or, HID.

However, because the protocol is self-describing, it’s ideal for supporting a larger universe of disparate devices. (By self describing, we mean that a HID device announces its: purpose, capabilities, and packet sizes upon connection to a host PC.)

The motion sensor described in this paper is intended as a teaching tool—you can use this device, and its firmware, to learn about the HID protocol and the new WinRT API. This sensor is designed to work with the **HidInfraredSensor** app that ships as a sample in the modern SDK.

If you’re new to the HID protocol, or the USB transport, a great starting point is Jan Axelson’s book “USB Complete”. Read the sections on HID before you dive into the details of the firmware. Another helpful source of information is the Developers forum on [usb.org](http://www.usb.org/developers).

# Quickstart

This section of the whitepaper is designed to help you quickly build and deploy the motion sensor. This sensor is designed to work with the sample **HidInfraredSensor** app that ships on the MSDN Samples Gallery for Windows 8.1. This sample monitors, or “listens” for, motion events from the sensor. When an event occurs, and motion is detected, the app captures a brief video using the built-in web cam on the device.

The app monitors events using the new [HID WinRT API](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.aspx); it captures video using the [MediaCapture API](http://msdn.microsoft.com/en-us/library/windows/apps/windows.media.capture.aspx).

(For a more detailed look at the inner workings of the device, see [The device firmware](#_The_device_firmware) later in this document.)

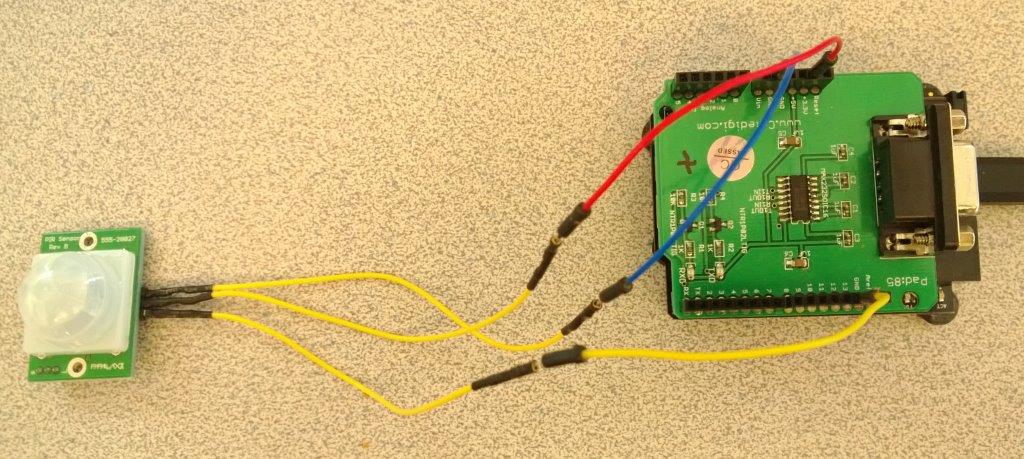
## Hardware Parts List

In order to build the complete motion-sensor, you’ll need the following parts:

|  |  |  |
| --- | --- | --- |
| Part | Description | Online sites with available parts |
| Netduino Plus | Development board with programmable microcontroller that supports the .Net Micro Framework. | [Netduino](http://www.netduino.com/netduinoplus2/specs.htm) |
| RS232 shield | RS232 module used to download and debug the firmware. (This shield is required for the beta version of the manufacturer’s firmware that we’re using.) | [CuteDigi](http://www.cutedigi.com/arduino-shields/rs232-shield-for-arduino.html#googlebase) |
| Passive Infrared motion sensor | Simple passive-infrared motion sensor that outputs a high signal when it detects motion. | [Parallax](http://www.parallax.com/tabid/768/productid/83/default.aspx) |
| RS232-to-USB converter cable | Cable used for downloading motion-sensor firmware via the RS232 shield. (Note that an FTDI chipset is required for compatibility with the shield.) | [Parallax](http://www.parallax.com/Store/Accessories/CablesConverters/tabid/166/CategoryID/40/List/0/SortField/0/Level/a/ProductID/378/Default.aspx) |
| Power supply | 9V 650mA power supply for the Netduino board. | [Amazon](http://www.amazon.com/Wall-Adapter-Power-Supply-650mA/dp/B003XZSZWO/ref=pd_bxgy_pc_text_y) |
| USB to micro-usb cable | Cable used for sending HID packets from the Netduino to your Windows 8.1 tablet or laptop | [Amazon](http://www.amazon.com/AmazonBasics-USB-Cable-Micro-Meters/dp/B003ES5ZSW/ref=sr_1_3?ie=UTF8&qid=1375128672&sr=8-3&keywords=USB+to+micro-USB+cable) |

A number of the circuit parts can be ordered from sources other than those listed in the table. However, for some parts, like the RS232-to-USB converter, you may find that other manufacturer’s devices are incompatible with the Netduino.

The following image shows the complete motion-sensor circuit (the RS232 shield is attached to the Netduino Plus):



## Preparing your Netduino Plus

When your Netduino Plus arrives from the manufacturer, the firmware that’s installed does not support the HID protocol. You’ll need to configure your development board by installing version 4.1.1 of the beta firmware which includes support for HID. This firmware is available on the Secret Labs website. You’ll find a zip folder containing the beta firmware [here](http://forums.netduino.com/index.php?/topic/1593-netduino-plus-firmware-v411-beta-1/).

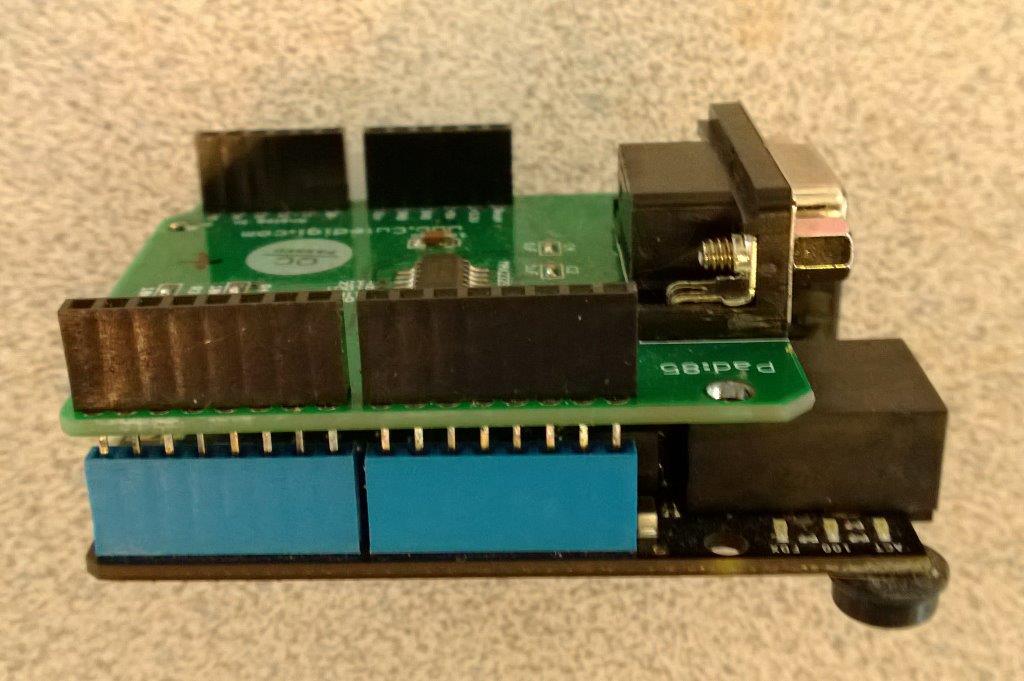
The download page on the website includes instructions for updating the firmware. However, these instructions are fairly complex--particularly if you’re new to the Netduino. This [video](https://www.youtube.com/watch?v=RkjAmrXIRuo) on Youtube to is a helpful, and concise, description of the firmware upgrade process.

## Building the circuit

After you’ve upgraded the firmware on your board, you’re ready to begin constructing the motion-sensor circuit.

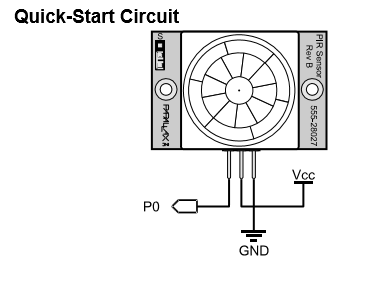
### Attaching the RS232 Shield

The first step requires you to attach the RS232 shield to your board. (The Netduino is pin-compatible with the Arduino; so, if you’ve been working with the Arduino and have an RS232 shield handy, you can use it.) Snap the RS232 shield onto the Netduino Plus as shown in this illustration:



### Attaching the Passive Infrared sensor

After you’ve attached the RS232 shield, the next step is to attach the passive infrared sensor to a 3.3V power source, ground, and pin 13 of the Netduino Plus. The following image from the Parallax datasheet for the sensor shows the pin-outs:



You’ll attach the pin labeled Vcc in this illustration to the 3.3V pin on the RS232 shield. You’ll attach the pin labeled GND to the corresponding Ground pin on the shield. And, you’ll attach the pin labeled P0 to pin 13 on the shield. When you’re done, the complete circuit look similar to the illustration in the section entitled [Hardware Parts List](#_Hardware_Parts_List).

## Installing the device firmware

There are two layers, or instances, of firmware on the Netduino Plus. The first is the manufacturer’s firmware; the second is your device firmware. The manufacturer’s firmware processes the requests from your device firmware. The manufacturer’s firmware is loaded once onto the development board and then used each time you power-up the device. On the other hand, you will refresh your device firmware repeatedly during your development and prototyping process.

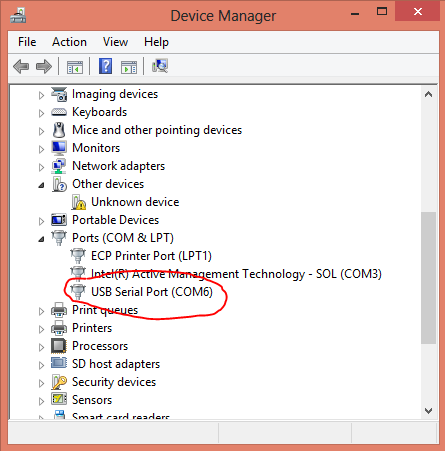
In order to install any device firmware, you’ll first need to install an instance of Visual C# Express 2010 on your development machine. You’ll find a link to the download [here](http://www.netduino.com/downloads/).

For most Netduino Plus projects, you can download and debug your firmware using the native USB connection. However, the beta version of the manufacturer’s firmware requires an RS232 connection for downloading and debugging your firmware. (This is why the RS232 shield is required.)

### Identifying the COM port

After you’ve installed an instance of Visual C# Express, you’ll want to attach the RS232-to-USB cable and open Windows Device Manager to determine which COM port Windows assigned to that cable.

After attaching the Parallax RS232 to USB converter to my development machine, Windows mapped it to COM6:

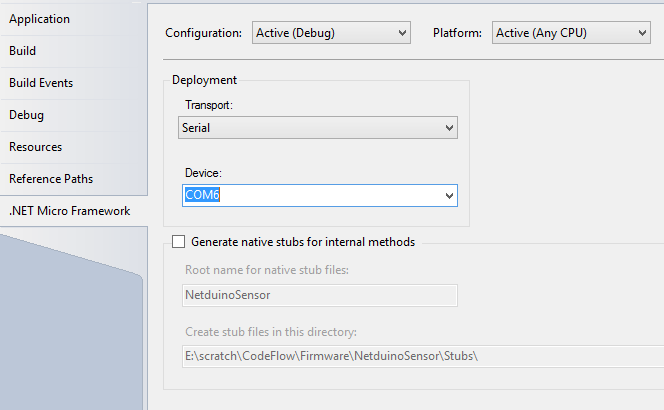


### Completing the installation

Now that we know the COM port associated with the converter, we can power-up our Netduino Plus, attach the RS232-to-USB cable, and start an instance of Visual C# Express to complete the download.

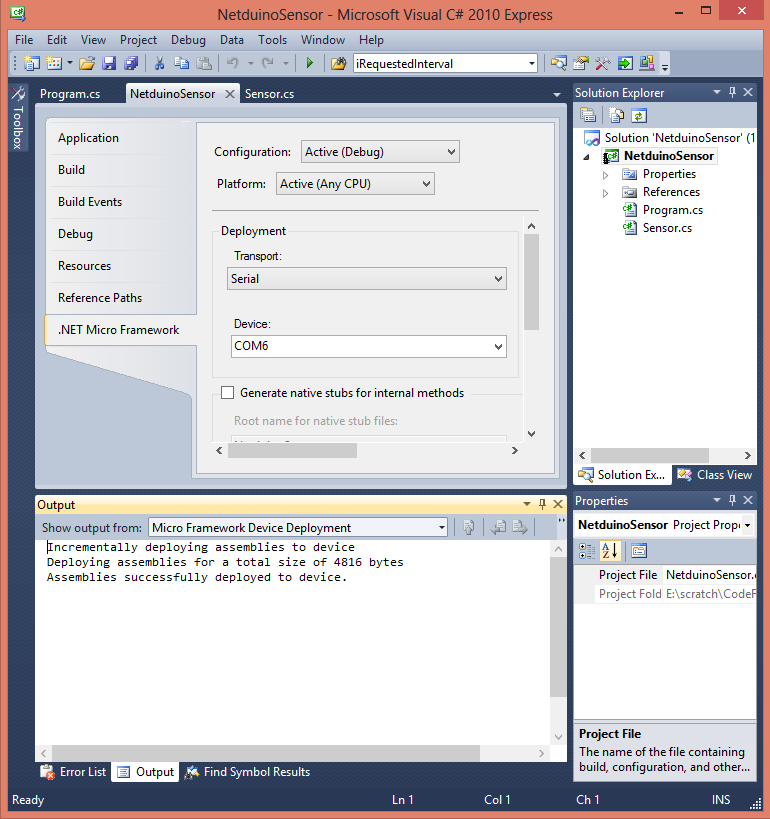
The first thing we’ll do, after starting Visual C# Express, is to identify the correct transport and COM port. You’ll do this by right clicking on the project name in the Solution Explorer pane and choosing the Properties menu.

Once the Properties dialog appears, you’ll choose the .NET Micro Framework tab and make the necessary selections:



After specifying the Transport and Device, you can deploy the firmware. Again, right click the project name in the Solution Explorer pane and, this time, choose Deploy.

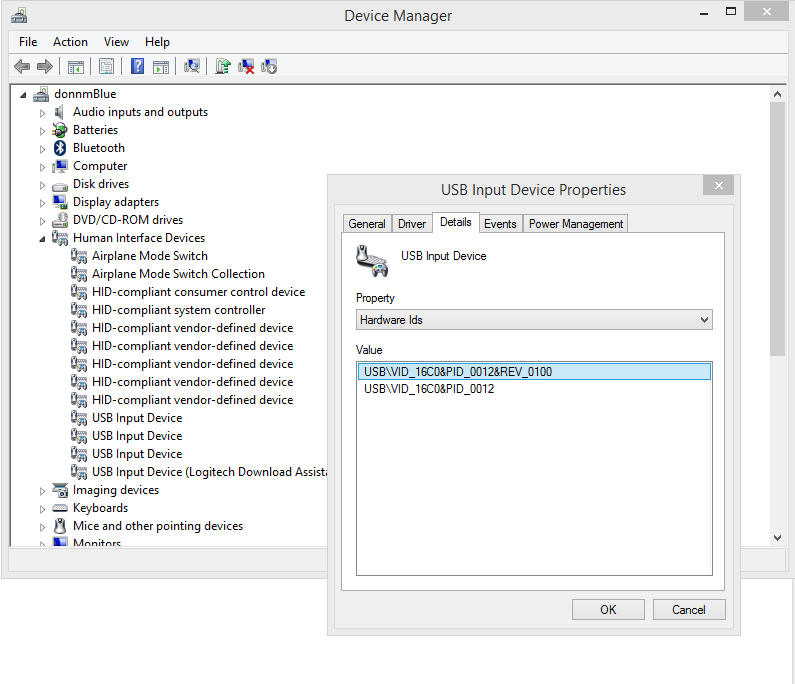
When the deployment completes, Visual C# Express will report the success in the Output pane:



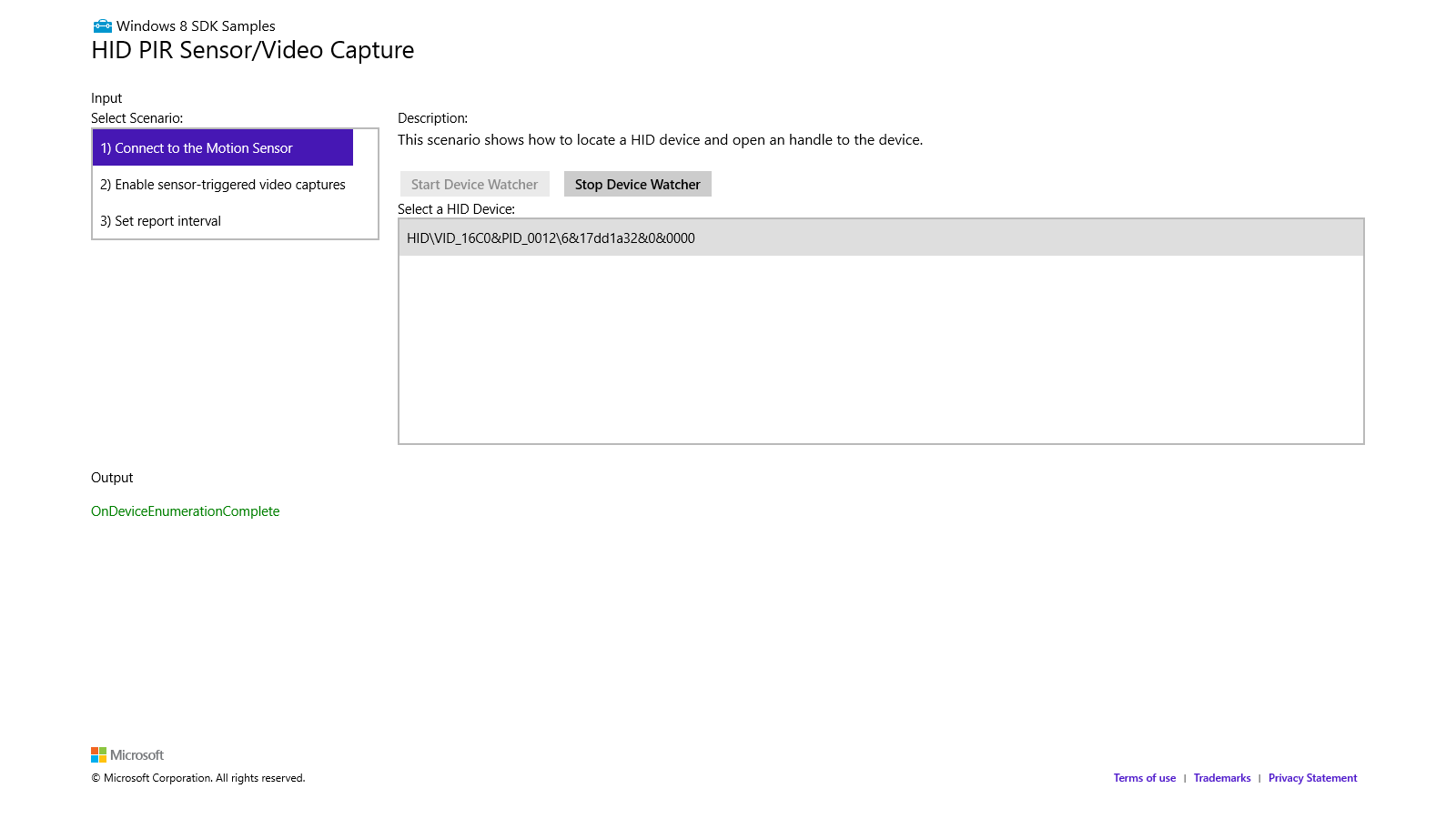
You’re now ready to attach your device to a Windows 8.1 tablet or laptop and test it with the **HidInfraredSensor** sample app.

You’ll want to detach the RS232 cable, power your Netduino down, and then restart it with the auxiliary power supply. Give the device several seconds to power-up and then attach the USB cable to the Netduino. After doing this, you should see your device added to the collection of HID devices in Device Manager.

You can use the Properties dialog of the Device Manager to examine the Hardware Ids of each USB Device that appear in the dialog. If Windows installed the motion sensor, you’ll see a VID of 0x16C and a PID of 0x12 as Hardware Id values for your new device.



Once the device is installed, you can install and build the sample app. After the app starts, you can select the motion sensor and begin testing video captures.



# The device firmware

This section of the whitepaper is a detailed look at the device firmware for the motion sensor. At the outset, we want to thank the folks at Secret Labs, and the forum contributors, for the work they’ve done to support HID over USB on the Netduino platform. The starting point for this firmware was a sample on the forum; it’s the UsbHidEchoNetduinoApp that’s found [here](http://forums.netduino.com/index.php?/topic/1514-driverless-pc-netduino-communication-using-usb/). In addition, a sample [class](http://forums.netduino.com/index.php?/topic/1246-parallax-pir-sensor-class/) for the PIR, found on the forum, provided a starting point for the integration of this sensor.

## Supporting the USB transport

Microsoft supports HID devices running over the following transports:

* USB
* Bluetooth
* Bluetooth LE
* I2C

However, the sample device described in this paper only supports the USB transport. (For more information about the other transports, refer to the MSDN [library](http://msdn.microsoft.com/en-us/library/windows/hardware/jj127210(v=vs.85).aspx).)

The phrase “USB transport” refers to the fact that the USB drivers are tasked with transporting packets back and forth between the device and the HID driver, or drivers. Unlike the HID drivers which contain logic to process the packets, the USB drivers simply move them. The USB drivers will move packets in both directions: packets originating with the device are passed up to the HID driver (which passes them on to the API if there are interested apps); packets originating with the HID driver are passed back down to the device.

Windows uses specific data issued by the device upon connection to identify which USB drivers it should load. Among this data are the Vendor ID (VID) and the Product ID (PID).

### Defining the Sensor class

The firmware for the motion-sensor device is built around three classes:

|  |  |
| --- | --- |
| Class name | Description |
| PIR | Integrates the passive-infrared sensor with the firmware. |
| Program | Supports a single Main routine which is invoked at startup. |
| Sensor | Defines the USB and HID settings for the motion sensor. In addition, it supports the methods which send input reports and read output reports. |

The Sensor class contains all of the code required to configure the USB transport. This includes the code that:

* Configures a read endpoint
* Configures a write endpoint
* Specifies the Vendor ID (VID)
* Specifies the Product ID (PID)
* Specifies friendly names (manufacturer name, product name, etc.)
* Specifies other, required USB settings for a HID device.

Most of the USB configuration code is found in the **ConfigureHID** method in the Sensors.cs module. This method, in turn, creates and initializes a [Configuration](http://msdn.microsoft.com/en-us/library/microsoft.spot.hardware.usbclient.configuration(v=vs.102).aspx) object which contains the device’s USB settings (endpoints, VID, PID, and so on).

The read endpoint allows the device to receive packets from the API and the HID driver. The write endpoint allows the driver to send packets up through the driver stack to the API.

Windows uses the Vendor ID, Product ID, and other USB settings (which were specified in the **ConfigureHID** method) to determine whether the device is a valid USB device and then to load the appropriate drivers.

### Opening the device connection

The Sensor class includes an **Open** method which is called from within the **Main** routine of the Program class. The **Open** method:

* Retrieves the available USB controllers
* Invokes the **ConfigureHID** method to establish the device’s USB and HID settings
* Invokes the **Start** method on the first available controller
* Creates a USB stream object with read and write endpoints

### Closing the device connection

The Sensor class includes a **Close** method which is called when the device is detached from the host laptop or tablet. This method, in turn, invokes the **UsbController.Stop** method.

## Supporting the HID protocol

The HID protocol is based on reports: feature reports, input reports, and output reports. Feature reports can be sent by either the host (i.e. a connected laptop or tablet), or, the device. Input reports are sent by the device to the host. Output reports are sent by the host to the device.

In the case of our sample motion sensor, the input report is a very simple two byte packet. The first byte indicates whether motion was detected (1 or 0); the second byte indicates the current report interval in seconds.

The output report, for the sample device, is even simpler—it’s a single byte that specifies the report interval.

While the USB drivers are responsible for simply moving packets back and forth, the HID driver is responsible for:

* Processing input reports and passing them along to any interested apps
* Processing output reports and sending them down to the device.

The new [HID WinRT API](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.aspx) for Windows 8.1 gives Windows Store apps the ability to easily access input reports and feature reports from a HID device. In addition, the new API gives Store apps the ability to send output reports to a connected HID device.

### Supporting the HID input report

The input report is defined as a structure in the Sensor.cs module:

struct InputReport

{

public byte Presence; // 1 if presence detected; 0 otherwise

public byte Interval; // report interval (or frequency) in seconds

}

The firmware issues input reports using the [UsbStream](http://msdn.microsoft.com/en-us/library/microsoft.spot.hardware.usbclient.usbstream(v=vs.102).aspx) object that it created in the **Open** method. These input reports are issued within the **SendInputReport** method when the firmware invokes the **stream.Write** method.

protected void SendInputReport(InputReport report)

{

byte[] inputReport = new byte[2];

inputReport[0] = (byte)report.Presence;

inputReport[1] = (byte)report.Interval;

stream.Write(inputReport, 0, 2);

}

### Supporting the HID output report

The output report is also defined as a structure in the Sensor.cs module:

struct OutputReport

{

public byte Interval; // report interval (or frequency) in seconds

}

The firmware receives output reports via the same UsbStream object that it created in the **Open** method. These output reports are received within the **GetOutputReport** method.

protected byte GetOutputReport()

{

byte[] outputReport = new byte[1];

int bytesRead = 0;

if (stream.CanRead)

{

bytesRead = stream.Read(outputReport, 0, 1);

}

if (bytesRead > 0)

return outputReport[0];

else

return 0;

}

### Creating the Report Descriptor

As I mentioned earlier, one of the features of a HID device is its self-reporting nature: Upon connecting to a host, the device provides the host with a description of its purpose, capabilities and packet-format in what’s called a report descriptor. This descriptor indicates where the device fits in the HID universe (Is it a mouse, a keyboard, a vendor-defined device, etc.?) And, the descriptor specifies the format of the individual feature reports, input reports, and output reports.

The report descriptor for the motion sensor is found in the file Sensors.cs:

hidGenericReportDescriptorPayload = new byte[]

{

0x06,0x55,0xFF, //HID\_USAGE\_PAGE\_VENDOR\_DEFINED

0x09,0xA5, //HID\_USAGE (vendor\_defined)

0xA1,0x01, //HID\_COLLECTION(Application),

// Input report (device-transmits)

0x09,0xA7, //HID\_USAGE (vendor\_defined)

0x15,0x00, //HID\_LOGICAL\_MIN\_8(0), // False = not present

0x25,0x01, //HID\_LOGICAL\_MAX\_8(1), // True = present

0x75,0x08, //HID\_REPORT\_SIZE(8),

0x95,0x01, //HID\_REPORT\_COUNT(1),

0x81,0x02, //HID\_INPUT(Data\_Var\_Abs),

0x09,0xA8, //HID\_USAGE (vendor\_defined)

0x15,0x01, //HID\_LOGICAL\_MIN\_8(1), // minimum 1-second

0x25,0x3C, //HID\_LOGICAL\_MAX\_8(60), // maximum 60-seconds

0x75,0x08, //HID\_REPORT\_SIZE(8),

0x95,0x01, //HID\_REPORT\_COUNT(1),

0x81,0x02, //HID\_INPUT(Data\_Var\_Abs),

// Output report (device-receives)

0x09,0xA9, //HID\_USAGE (vendor\_defined)

0x15,0x01, //HID\_LOGICAL\_MIN\_8(1), // minimum 1-second

0x25,0x3C, //HID\_LOGICAL\_MAX\_8(60), // maximum 60-seconds

0x75,0x08, //HID\_REPORT\_SIZE(8),

0x95,0x01, //HID\_REPORT\_COUNT(1),

0x91,0x02, //HID\_OUTPUT(Data\_Var\_Abs),

0xC0 //HID\_END\_COLLECTION

};

The first two lines of the descriptor inform the host that this particular device is vendor defined.

0x06,0x55,0xFF, //HID\_USAGE\_PAGE\_VENDOR\_DEFINED

0x09,0xA5, //HID\_USAGE (vendor\_defined)

The fourth through the fifteenth lines describe the format of the two-byte input-report. Lines four through nine describe the first byte of the input report which specifies the motion (or presence) data.

0x09,0xA7, //HID\_USAGE (vendor\_defined)

0x15,0x00, //HID\_LOGICAL\_MIN\_8(0), // False = not present

0x25,0x01, //HID\_LOGICAL\_MAX\_8(1), // True = present

0x75,0x08, //HID\_REPORT\_SIZE(8),

0x95,0x01, //HID\_REPORT\_COUNT(1),

0x81,0x02, //HID\_INPUT(Data\_Var\_Abs),

The tenth through fifteenth lines describe the second byte of the input report which specifies the report interval (in seconds).

0x09,0xA8, //HID\_USAGE (vendor\_defined)

0x15,0x01, //HID\_LOGICAL\_MIN\_8(1), // minimum 1-second

0x25,0x3C, //HID\_LOGICAL\_MAX\_8(60), // maximum 60-seconds

0x75,0x08, //HID\_REPORT\_SIZE(8),

0x95,0x01, //HID\_REPORT\_COUNT(1),

0x81,0x02, //HID\_INPUT(Data\_Var\_Abs),

The sixteenth through twenty-first lines describe the format of the single-byte output report.

0x09,0xA9, //HID\_USAGE (vendor\_defined)

0x15,0x01, //HID\_LOGICAL\_MIN\_8(1), // minimum 1-second

0x25,0x3C, //HID\_LOGICAL\_MAX\_8(60), // maximum 60-seconds

0x75,0x08, //HID\_REPORT\_SIZE(8),

0x95,0x01, //HID\_REPORT\_COUNT(1),

0x91,0x02, //HID\_OUTPUT(Data\_Var\_Abs),

The report descriptor for the sample device is included as part of the [UsbController.Configuration](http://msdn.microsoft.com/en-us/library/ee432010.aspx) object that’s created within the **ConfigureHID** method in Sensor.cs.

## Integrating the motion sensor

The passive infrared (PIR) sensor that is used as a motion-sensor for this solution is represented by a simple class that was found [here](http://forums.netduino.com/index.php?/topic/1246-parallax-pir-sensor-class/) on the Netduino forums.

The only change, or modification, was to replace the first two event-handler arguments (*data1* and *data2*) with more descriptive argument names (*port* and *motion*). The definition for this class is found in the file Sensor.cs.

public class PIR{

private InterruptPort \_sensor;

public event PIRTriggeredEventHandler SensorTriggered;

/// <summary>

/// Initializes a new instance of the PIR class.

/// </summary>

/// <param name="portId">The port id.</param>

public PIR(Cpu.Pin portId){

\_sensor = new InterruptPort(

portId,

false,

Port.ResistorMode.Disabled,

Port.InterruptMode.InterruptEdgeBoth);

\_sensor.OnInterrupt += new NativeEventHandler(

(port, motion, time) =>

{

OnSensorTriggered(port, motion, time);

}

);

}

The PIR class creates an [InterruptPort](http://msdn.microsoft.com/en-us/library/cc532335.aspx) object that allows the firmware to “listen” for input from the Netduino board. The first argument to the **InterruptPort** constructor, *portId*, identifies the pin that we’re monitoring. The second argument, *glitchFilter*, specifies whether we’ve enabled a “glitch” filter, the third argument, *ResistorMode*, specifies that the resistor mode is disabled, and, the fourth argument, *InterruptMode*, specifies that the firmware should monitor events when the pin goes either high or low.

Since our use of this sensor is only concerned with the state of the *motion* variable, we ignore the *port* and *time* arguments.

### Sensor events

The event handler (that corresponds to the PIR.SensorTriggered event) is found in the Main routine in the Program.cs file.

int State = 0;

…

PIR.SensorTriggered += new PIRTriggeredEventHandler(

(triggered, time) =>

{

// Each time the PIR toggles state

// we set the State variable

State = (int)triggered;

}

);

Each time the sensor triggers an event—that is, each time the monitored pin goes high or low, we set the *State* variable.

while (true)

{

// We invoke the Update method every CurrentInterval seconds

// and pass the current value of State

RequestedInterval = sensor.Update(State, CurrentInterval);

// Check for a possible new interval requested via an

// output report.

if (RequestedInterval != 0)

{

NetduinoInterval = RequestedInterval \* 1000;

CurrentInterval = RequestedInterval;

}

else

NetduinoInterval = CurrentInterval \* 1000;

// Toggle the LED every iNetuinoInterval seconds

led.Write(true);

Thread.Sleep(NetduinoInterval);

led.Write(false);

}

The *State* variable is passed as an argument to the **Sensor.Update** method. And, it’s passed at a frequency of the current report interval. The **Update** method, in turn, packages this value into an input report and sends that report to the host. (Refer to the **Update** and **SendInputReport** methods found in Sensor.cs.)

### The report interval

The firmware supports a report interval specified in units of seconds. An app requests a new interval by sending an output report to the device. The device, in turn, reports the current interval in each input report that it sends to any connected app.

The firmware applies the current interval by invoking the [Thread.Sleep](http://msdn.microsoft.com/en-us/library/ee436250.aspx) method for the number of seconds specified by the current interval.

led.Write(true);

Thread.Sleep(NetduinoInterval);

led.Write(false);

By pausing the While loop for this duration, registered apps receive input reports at the specified interval.

# Resources

HID Information on USB.org

<http://www.usb.org/developers/hidpage/>

Netduino Plus technical specifications

<http://www.netduino.com/netduinoplus/specs.htm>

HID WinRT API

<http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.aspx>

.NET Micro Framework API

<http://msdn.microsoft.com/en-us/library/ee435793.aspx>