HidInfraredSensor App

August 6, 2013

Abstract

This paper describes how to write an application that monitors a motion sensor and triggers a video capture when motion is detected. The app uses the new HumanInterfaceDevice (HID) API to monitor the sensor and the MediaCapture API to create the video. It assumes that the reader is familiar with the store apps and the HID protocol.

This information applies to the following operating systems:   
 Windows 8.1

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

The current version of this paper is maintained on the Web at:   
 <http://www.microsoft.com/> <**Writer: Get correct URL or delete this paragraph if not applicable.**>

**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.

**<The following sentence may be omitted if there are no examples using names or identifiers:>** Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

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. **<insert the following sentence if you will allow modification:** You may modify this document for your internal, reference purposes.

© 2013 Microsoft. All rights reserved.



Contents

[The HID infrared-sensor app 3](#_Toc363559374)

[Supporting the device-connect scenario 4](#_Toc363559375)

[Establishing a device connection 4](#_Toc363559376)

[Supporting the device watcher 8](#_Toc363559377)

[Handling device disconnect and reconnect 8](#_Toc363559378)

[Supporting the video-capture scenario 9](#_Toc363559379)

[Monitoring the sensor input reports 10](#_Toc363559380)

[Supporting video capture 11](#_Toc363559381)

[Starting and stopping the capture 12](#_Toc363559382)

[Supporting the set report-interval scenario 13](#_Toc363559383)

[Resources 14](#_Toc363559384)

# The HID infrared-sensor app

This sample demonstrates the power of the new [HumanInterfaceDevice](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.aspx) (HID) WinRT API for Windows 8.1. The new API allows apps to retrieve data from, as well as control, a HID device. The sample app is designed to work with an attached infrared motion-sensor that detects movement. (For information about building this device, refer to the “Building a HID Motion Sensor” paper.) The app monitors the motion sensor for movement, and, once detected, captures five seconds of video.

The video is captured using the default camera on the Windows PC that’s running the app. The capture is accomplished with the [MediaCapture](http://msdn.microsoft.com/en-us/library/windows/apps/windows.media.capture.aspx) API.

The app supports three “scenarios”, and, each scenario maps to specific features in the app’s UI. In addition, each scenario maps to a corresponding XAML and CS source file. The following table lists each scenario, its purpose, and the corresponding modules.

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Purpose** | **Corresponding Modules** |
| Device Connect | Supports all aspects of connecting a HID device to a Windows 8.1 PC.  Enumerates the connected infrared sensors so that the user can select one.  Establishes a device-watcher which monitors the status of the device. (The device-watcher fires an event when the user disconnects or reconnects the selected HID device.) | Scenario1\_DeviceConnect.xaml  Scenario1\_DeviceConnect.xaml.cs |
| Sensor-Triggered Video Capture | Monitors the selected motion-sensor.  If the user has enabled video-captures, this module contains code that captures five-seconds of video each time motion is detected. | Scenario2\_SensorTriggeredVideoCapture.xaml  Scenario2\_SensorTriggeredVideoCapture.xaml.cs |
| Set Report Interval | Allows the user to control the frequency at which the motion-sensor reports its status. (The default interval is 1 second; but, the user can choose intervals from 1 to 5 seconds.) | Scenario3\_SetReportInterval.xaml  Scenario3\_SetReportInterval.xaml.cs |

## Supporting the device-connect scenario

The device-connect scenario supports several aspects of connecting a HID device to a Windows 8.1 PC. This includes:

* Enumerating connected devices (so that the user can select one)
* Establishing a device-watcher for the connected device
* Handling disconnection of the device
* Handling reconnection of the device

Note that an ideal example of device connection is demonstrated in the CustomHidDeviceAccess sample on MSDN. (The HidInfraredSensor sample focuses on combining the HID and MediaCapture APIs.)

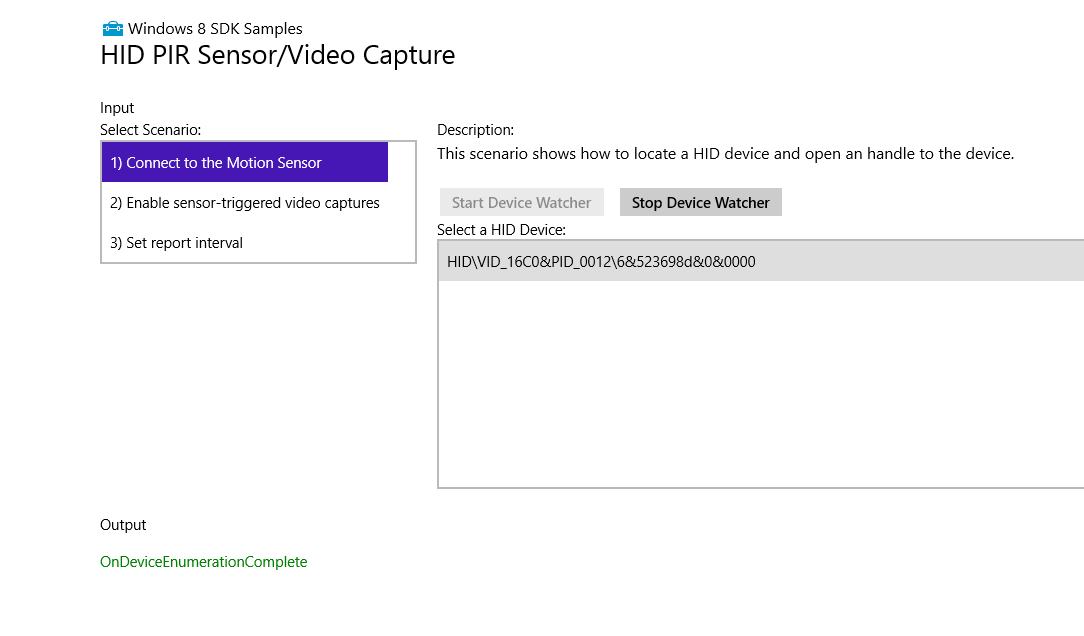
### Establishing a device connection

The code that handles the device connection is found in three modules: Scenario1\_DeviceConnect.xaml.cs, EventHandlerForDevices.cs, and DeviceList.cs. (The first module contains the primary code for this scenario; the latter two contain supporting functionality.)

The first phase of the connection occurs before the UI is visible. The second phase occurs after the UI is displayed and the user is able to choose a specific device from the connected HID devices. The app displays a DeviceInstanceId string for each connected device that includes the Vendor ID (VID) and Product ID (PID) for the given device. In the case of the sample motion-sensor, this string has the form:

HID\VID\_16C0&PID\_0012\6&523698d&0&0000

The following screenshot shows the app as it appears immediately after enumeration completed.



#### The first stage of device connection

The following table outlines the methods which are called during the first stage of device connection (before the UI is displayed). In addition, the table lists the tasks accomplished by each method.

|  |  |  |
| --- | --- | --- |
| **Module** | **Method** | **Notes** |
| Scenario1\_DeviceConnect.xaml.cs | **DeviceConnect** | Invokes the **HidInfraredSensor.InitializeComponent** method which initializes the app’s UI components such as the text blocks and buttons. (Note that although these components are initialized, they aren’t displayed until the **SelectDeviceInList** method completes.) |
| Scenario1\_DeviceConnect.xaml.cs | **InitializeDeviceWatchers** | Invokes the **HidDevice.GetDeviceSelector** method to retrieve a device selector string. (The selector is required in order to create a device watcher.)  Once the selector is obtained, the app invokes **DeviceInformation.CreateWatcher** to create the DeviceWatcher object and then **EventHandlerForDevice.Current.AddDeviceWatcher**. (This last method allows the app to monitor changes in device status.) |
| EventHandlerForDevices.cs | **AddDeviceWatcher** | Creates the event handlers for three device events:  1) Enumeration Completed  2) Device Added  3) Device Removed |
| Scenario1\_DeviceConnect.xaml.cs | **SelectDeviceInList** | This method checks to see if the user has selected a device, and, if so, it saves the index for that device. |

In terms of the HID API, the primary code of interest is found in the **InitializeDeviceWatchers** method. This code, in turn, invokes the [HidDevice.GetDeviceSelector](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.hiddevice.getdeviceselector.aspx) method and passes the *UsagePage* and *UsageId* for the motion sensor:

private void InitalizeDeviceWatchers()

{

// IR\_Sensor

var IR\_SensorSelector = HidDevice.GetDeviceSelector(IR\_Sensor.Device.UsagePage, IR\_Sensor.Device.UsageId);

// Create a device watcher to look for instances of the OSRFX2 device interface and IR\_Sensor device

var IR\_SensorWatcher = DeviceInformation.CreateWatcher(IR\_SensorSelector);

// Allow the EventHandlerForDevice to handle device watcher events that relates or effects our device (i.e. device removal, addition, app suspension/resume)

EventHandlerForDevice.Current.AddDeviceWatcher(IR\_SensorWatcher);

}

The *UsagePage* and *UsageId* values are defined in the file constants.cs:

public class IR\_Sensor

{

public class Device

{

public const UInt16 Vid = 0x16C0;

public const UInt16 Pid = 0x0012;

public const UInt16 UsagePage = 0xFF55;

public const UInt16 UsageId = 0xA5;

}

}

These class members correspond to values specified in the HID report descriptor that is defined in the device’s firmware:

hidGenericReportDescriptorPayload = new byte[]

{

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

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

…

#### The second stage of device connection

The second stage of device connection allows the user to make a selection from the list of connected devices. This stage establishes the currently selected device.

|  |  |  |  |
| --- | --- | --- | --- |
| **Module** | **Method** | **Notes** |  |
| EventHandlerForDevices.cs | **OnDeviceEnumerationComplete** | Notifies the user that device enumeration has completed by writing a “OnDeviceEnumerationComplete” string to the Output section of the app’s window. |  |
| DeviceList.cs | **AddDeviceToList** | Updates the internal list of devices. |  |
| Scenario1\_DeviceConnect.xaml.cs | **ConnectDevices\_SelectChanged** | Invoked when the user selects a device from the “Select a HID Device” list in the app’s window. |  |
|  | **OpenDevice** | Invoked by the **SelectChanged** method. This method, in turn, invokes the [HidDevice.FromIdAsync](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.hiddevice.fromidasync.aspx) method from the HID API. |  |
| DeviceList.cs | **SetCurrentDevice** | Invoked by **OpenDevice**. This method saves the new device ID as a global object so that it can be accessed from the other scenarios. |  |

In terms of the HID API, the primary code of interest is found in the **OpenDevice** method. This code, in turn, invokes the [HidDevice.FromIdAsync](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.hiddevice.fromidasync.aspx) method which returns a [HidDevice](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.hiddevice.aspx) object that the app uses to: access the device, retrieve input reports, and send output reports.

public static async void OpenDevice(String id)

{

// It is important that the FromIdAsync call is made on the UI thread because the consent prompt can only be displayed

// on the UI thread.

await MainPage.Current.Dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () =>

{

HidDevice device = await HidDevice.FromIdAsync(id, FileAccessMode.ReadWrite);

// Save the device from the task to a global object so that it can be used later from other scenarios

DeviceList.Current.SetCurrentDevice(id, device);

MainPage.Current.NotifyUser("Device " + id + " opened", NotifyType.StatusMessage);

});

}

### Supporting the device watcher

Device enumeration occurs when the app is first started and begins even before the UI is displayed. After enumeration completes, the app monitors device status.

Device status is reported by a [DeviceWatcher](http://msdn.microsoft.com/library/windows/apps/br225446) object. As the name implies, this object “watches” the connected devices—if the user removes, or connects, their device the watcher reports the event to the app. (These events are only reported after the enumeration process is finished.)

The code that supports watcher is found in three separate modules:

|  |  |
| --- | --- |
| **Module** | **Purpose** |
| Scenario1\_DeviceConnect.xaml.cs | Displays the DeviceWatcher UI (two buttons which let the user start or stop the DeviceWatcher). Initializes the DeviceWatcher. Renders the current device status in the app’s window. |
| EventHandlerForDevices.cs | Supports the enumeration and DeviceWatcher events. A single enumeration event is fired when the enumeration completes. The DeviceWatcher events are fired when a device is removed or added. |
| DeviceList.cs | Maintains a list of connected devices as well as an identifier for the current device. |

### Handling device disconnect and reconnect

During the first phase of device connection (before the UI is displayed), the app invokes the **AddDeviceWatcher** method in the EventHandlerForDevices.cs module. This method establishes three event handlers:

1) Enumeration Completed

2) Device Added

3) Device Removed

Once the app establishes these event handlers, it can monitor the status of a connected device. In the case of device removal, or disconnection, the **OnDeviceRemoved** handler is invoked. This is followed by a series of additional method calls to remove the device from the internal devices list, close the device, reset the current device, and so on. The following table lists the sequence of methods as they are called when the user disconnects a device:

|  |  |  |
| --- | --- | --- |
| **Module** | **Method** | **Notes** |
| EventHandlerForDevices | **OnDeviceRemoved** | The device-watcher needs to be started before device-removal is tracked. |
| DeviceList.cs | **RemoveDeviceFromList** |  |
| Scenario1\_DeviceConnect.xaml.cs | **ConnectDevices\_SelectChanged** |  |
| Scenario1\_DeviceConnect.xaml.cs | **CloseDevice** |  |
| DeviceList.cs | **SetCurrentDevice** |  |
| Scenario1\_DeviceConnect.xaml.cs | **CloseDevice** | Issues “Device is Closed” message in the Output section of the app’s window. |

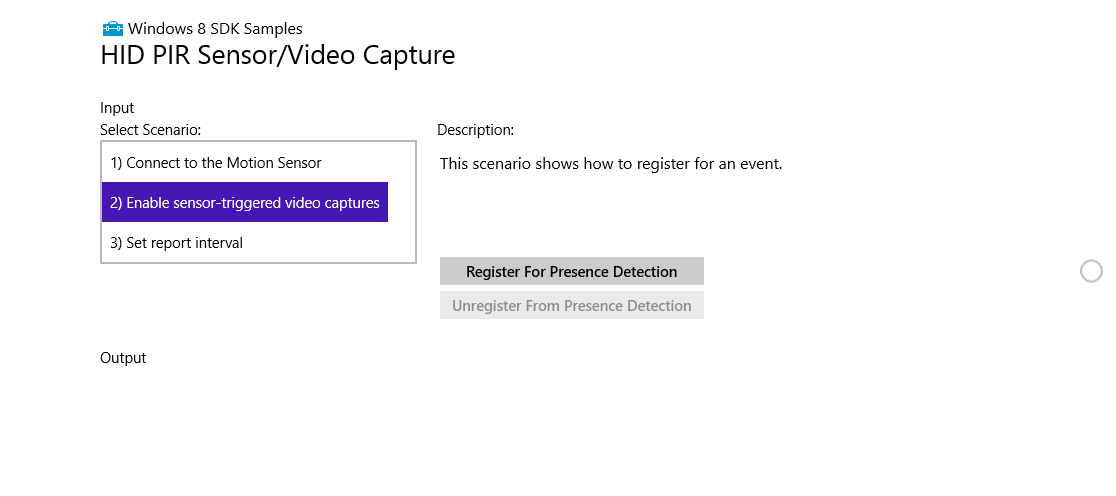
In the case of device connection, the **OnDeviceAdded** handler is invoked. This method issues an “OnDeviceAdded…” message to the Output section of the app’s window. After this, the internal list of devices is updated with the identifier for the added device.

|  |  |  |
| --- | --- | --- |
| **Module** | **Method** | **Notes** |
| EventHandlerForDevices | **OnDeviceAdded** | Issues “OnDeviceAdded:…” message to Output area of app’s window. |
| DeviceList.cs | **AddDeviceToList** |  |

## Supporting the video-capture scenario

The video-capture scenario monitors input reports issued by the motion-sensor and triggers a five-second video capture when motion is detected.

The primary workhorse of the video capture scenario is the **OnGeneralInterruptEvent** handler that is found in the Scenario2\_SensorTriggeredVideoCapture.xaml.cs module. The app registers this handler when the user chooses the “Enable sensor-triggered video capture” scenario and presses the “Register For Presence Detection” button.



Once registered, this handler examines each input report issued by the sensor, and if motion is detected, it starts the five-second video capture. The video results are stored in a file named “video.mp4”.

### Monitoring the sensor input reports

The sensor input reports are defined in the firmware as a 2-byte packet:

struct InputReport

{

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

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

}

The first byte specifies whether motion (or “presence”) was detected. This value is 1 if the sensor detected motion and 0 if not. The second byte specifies the current report interval in units of seconds. (When the device starts, the default report interval is 1 second.)

When the **OnGeneralInterruptEvent** handler is invoked, the second argument, *eventArgs*, contains the most recent input report.

private async void OnGeneralInterruptEvent(HidDevice sender, HidInputReportReceivedEventArgs eventArgs)

{

…

If you’re new to the HID protocol, you’d expect *eventArgs* to contain two bytes. However, the HID driver actually inserts a third byte at the beginning of each report. This byte specifies a report identifier. (Because our sample device supports a single input report, the identifier will always contain the value 0. And, the app ignores it.)

The first task accomplished by the event handler is to extract the three bytes of data from the input report and place this data in an array of bytes:

// Retrieve the sensor data

HidInputReport inputReport = eventArgs.Report;

IBuffer buffer = inputReport.Data;

DataReader dr = DataReader.FromBuffer(buffer);

byte[] bytes = new byte[inputReport.Data.Length];

dr.ReadBytes(bytes);

Once we have this data, we can examine the second byte in the array. (Remember, this byte contains a value of 1 if motion is detected; but, it set to 0 otherwise.)

// The first byte contains the motion data

if ((bytes[1] == 1) && !Capture)

{

Capture = true;

…

In addition to examining the motion data from the input report, we also examine the Capture flag to ensure that a video capture isn’t already underway.

### Supporting video capture

Video captures are enabled through the [MediaCapture](http://msdn.microsoft.com/en-us/library/windows/apps/windows.media.capture.mediacapture.aspx) API. With very little code (as demonstrated in the app), it’s possible to capture video and audio segments using the default webcam on a Windows 8.1 PC. The sample app captures a five-second video each time motion is detected; the resulting file, video.mp4, is stored in the \Videos folder of the device running the app.

#### Initializing the MediaCapture object

The preparatory work for the video capture is done in the **RegisterForInterruptEvent** method. (The app invokes this method when the user clicks the “Register For Presence Detection” button.) The media-capture code within this method:

1. Creates a new MediaCapture object
2. Initializes the object
3. Sets an event handler that Windows triggers if the video exceeds the limitations of the device.
4. Sets an event handler that Windows triggers if the capture fails.

private async void RegisterForInterruptEvent(TypedEventHandler<HidDevice, HidInputReportReceivedEventArgs> eventHandler)

{

if (interruptEventHandler == null)

{

// Save the interrupt handler so we can use it to unregister

interruptEventHandler = eventHandler;

DeviceList.Current.CurrentDevice.InputReportReceived += interruptEventHandler;

UpdateRegisterEventButton();

// Prepare for media captures

CaptureMgr = new Windows.Media.Capture.MediaCapture();

await CaptureMgr.InitializeAsync();

CaptureMgr.RecordLimitationExceeded += new Windows.Media.Capture.RecordLimitationExceededEventHandler(RecordLimitationExceeded); ;

CaptureMgr.Failed += new Windows.Media.Capture.MediaCaptureFailedEventHandler(Failed); ;

rootPage.NotifyUser("Video capture enabled." , NotifyType.StatusMessage);

}

}

The event handlers that deal with videos that exceed device limitations, or, failed captures are found early on in the Scenario2\_SensorTriggeredVideoCapture.xaml.cs file.

### Starting and stopping the capture

Once the user presses the “Register For Presence Detection” button and the app executes the **RegisterForInterruptEvent** method, the app begins processing input reports within the **OnGeneralInterruptEvent** method.

The capture begins when the second byte of an input report contains the value 1. The code which starts the video capture is found towards the end of **OnGeneralInterruptEvent**. The first time motion is detected, the app invokes [MediaCapture.StartRecordToStorageFileAsync](http://msdn.microsoft.com/en-us/library/windows/apps/Hh700863.aspx).

…

String fileName;

fileName = VIDEO\_FILE\_NAME;

m\_recordStorageFile = await Windows.Storage.KnownFolders.VideosLibrary.CreateFileAsync(fileName, Windows.Storage.CreationCollisionOption.GenerateUniqueName);

MediaEncodingProfile recordProfile = MediaEncodingProfile.CreateMp4(Windows.Media.MediaProperties.VideoEncodingQuality.Auto);

await m\_mediaCaptureMgr.StartRecordToStorageFileAsync(recordProfile, m\_recordStorageFile);

Since the device continues to send input reports regardless of whether a capture is underway, the app sets a *Capture* flag to true in order to indicate that a capture is started. This flag is necessary since the input reports may continue to report motion; but, we only want one video for a given series of positive reports.

Note: If you’re writing an app for a device that sends continuous data, rather than periodic data (like the sample device), multiple callbacks may trigger at the same time. In this case, a single variable isn’t adequate to prevent another concurrent thread from entering the conditional statement. Instead, a lock is required.

In addition to setting the *Capture* flag to prevent unwanted captures, we also need to address the potential for false captures when the Parallax PIR sensor is returning to its default state. (The device requires several seconds to return to its default state. And, during this time, it generates one or more false-positive readings.) To filter false positive readings, we’ve created a *delay* variable that specifies a delay period in seconds. We create a [ThreadPoolTimer](http://msdn.microsoft.com/en-us/library/windows/apps/windows.system.threading.threadpooltimer.aspx) to ensure that this *delay* transpires before we reset the *Capture* flag to false.

ThreadPoolTimer CapturePauseTimer = ThreadPoolTimer.CreateTimer(

async (source) =>

{

Capture = false;

await rootPage.Dispatcher.RunAsync(

CoreDispatcherPriority.Normal,

new DispatchedHandler(() =>

{

rootPage.NotifyUser("Presence sensor enabled.", NotifyType.StatusMessage);

}));

}, delay);

We use another ThreadPoolTimer to control the duration of the video capture. When this timer completes, we invoke the [MediaCapture.StopRecordAsync](http://msdn.microsoft.com/en-us/library/windows/apps/windows.media.capture.mediacapture.stoprecordasync.aspx) method. (The duration of this timer is specified by the *length* variable.)

ThreadPoolTimer VideoStopTimer = ThreadPoolTimer.CreateTimer(

async (source) =>

{

await m\_mediaCaptureMgr.StopRecordAsync();

await rootPage.Dispatcher.RunAsync(

CoreDispatcherPriority.Normal,

new DispatchedHandler(() =>

{

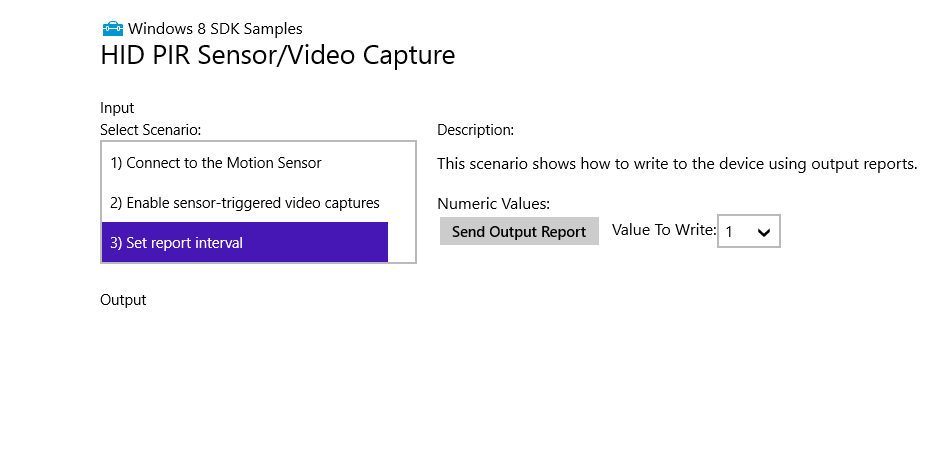
rootPage.NotifyUser("Video capture concluded.", NotifyType.StatusMessage);

}));

}, length);

## Supporting the set report-interval scenario

The report-interval scenario sends an output report to the motion-sensor and writes the count of bytes as well as the value written to the Output area of the app’s window. The app sends an output report after the user chooses the “Set report interval” scenario, selects a value from the “Value To Write” drop down, and then presses the “Send Output Report” button.



The primary method of the report-interval scenario is the **SendNumericOutputReportAsync** method that is found in the Scenario3\_SetReportInterval.xaml.cs module. This method, in turn, invokes the [HidDevice.SendOutputReportAsync](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.humaninterfacedevice.hiddevice.sendoutputreportasync.aspx) method to send an output report to the device.

private async Task SendNumericOutputReportAsync(Byte valueToWrite)

{

var outputReport = DeviceList.Current.CurrentDevice.CreateOutputReport();

Byte[] bytesToCopy = new Byte[1];

bytesToCopy[0] = valueToWrite;

WindowsRuntimeBufferExtensions.CopyTo(bytesToCopy, 0, outputReport.Data, 1, 1);

uint bytesWritten = await DeviceList.Current.CurrentDevice.SendOutputReportAsync(outputReport);

rootPage.NotifyUser("Bytes written: " + bytesWritten.ToString() + "; Value Written: " + valueToWrite.ToString(), NotifyType.StatusMessage);

}

The device firmware processes the output report (which contains a requested report interval) and uses this value to set the frequency at which it issues input reports back to the host.

# Resources

HumanInterfaceDevice (HID) API

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

HID Information on USB.org

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

MediaCapture API

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