# Connecting to USB Devices

During Build 2013 there was a great presentation on connecting to USB peripherals using the new Windows 8.1 RT APIs. In Windows 8 communicating with USB devices in a Store App (RT App) was not available. You could still connect to USB devices in desktop mode.

In this article I write about my journey to USB connectivity while recreating the session at build with a few code samples.

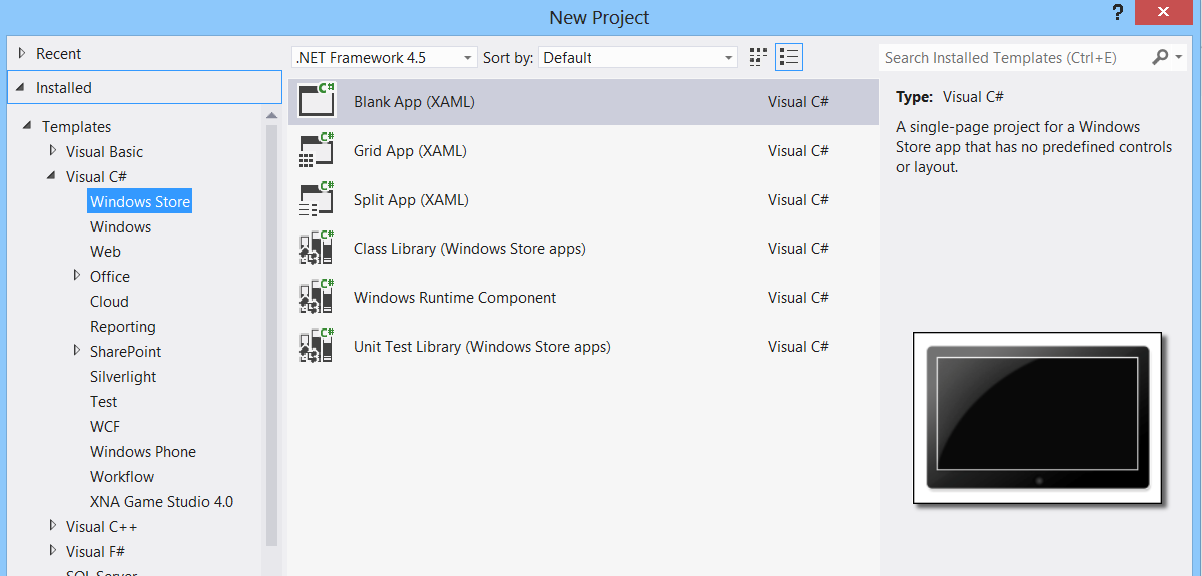
# Choosing a USB Device

For this blog I chose to use the Dream Cheeky Thunder USB Missile Launcher pictured below.



# Project Setup

First I created a very simple C# Windows Store application to communicate with the missile launcher.



In the XAML I added the follow button:

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

<Button Click="Button\_Connect">Connect</Button>

</Grid>

I added only the Connect button so I could check the connectivity of my new USB launcher.

In the code behind add the following event handlers for the click events for the buttons:

private void Button\_Connect(object sender, RoutedEventArgs e)

{

}

Note: I know – where’s the MVVM, the commands, etc. This exercise is about USB device connectivity.

# Connecting to the Launcher

Plug the USB launcher into the table. Great, done. Well, not quite. In order to allow access to the device the application will need to define it in the application manifest. At the time of this writing, users will need to add the device capability manifest entries in the XML as follows:

<Capabilities>

<m2:DeviceCapability Name="humaninterfacedevice">

<m2:Device Id="vidpid:2123 1010">

<m2:Function Type="usage:0010 0001" />

</m2:Device>

</m2:DeviceCapability>

</Capabilities>

The capability defines which device we are trying to load, and how we are going to use it.

|  |  |
| --- | --- |
| <m2:DeviceCapability Name="humaninterfacedevice"> | This describes to the app that we will be connecting to a HID device. |
| <m2:Device Id="vidpid: <vid> <pid>"> | This defines that we will connect to a specific device via “vidpid: xxxx xxxx”. I got this value from Device Manager. The manufacturer should be able to give you these values. |
| <m2:Function Type="usage:0010 0001"/>  <m2:Function Type="usage:<usagepage> <usageid>"/> | This defines a page within the USB device. Essentially this defines a bunch of communication protocols that the HID and USB DLL will use to talk with each other.  These values will need to be provided by the manufacturer.  There is a way to derive these in desktop mode – which I will not go into – in this blog.  For our Thunder launcher, the values are: UsagePage: 0x0010  UsageID: 0x0001 |

# Opening the USB Device

In this section we will connect to the USB device. Have a look at the following code:

|  |
| --- |
| //Create AQS - Advanced Query Syntax String.  //This is used to search for a device.  string aqs = HidDevice.GetDeviceSelector(usagePage, usageId);//, vendorId, deviceId);  //Find a device with a specific AQS.  //Leaving out the aqs will find all HID and USB devices  var myDevices = await Windows.Devices.Enumeration.DeviceInformation.FindAllAsync(aqs);  deviceInformation = myDevices[0];  hidDevice = await HidDevice.FromIdAsync(deviceInformation.Id, FileAccessMode.ReadWrite);  if (hidDevice == null)  {  //do some error handling  } |

I created a query string – Advanced Query String. This will help us identify the HID device based on the usagePage and usageID.

When I used the API DeviceInformation.FindAllAsync( aqs ), I should have been presented with a complete list of all HID devices with the usagePage and usageId.

I could then in theory use the device information returned from the query and open a HID device using:

HidDevice.FromIdAsync( <deviceID>, <FileAccessMode> );

I added the above code to the Connect click handler:

private void Button\_Connect(object sender, RoutedEventArgs e)

When I tested this –at this point, I received a valid Advance Query String. When enumerating the HID devices I did not receive any valid devices which I could interface with back in the enumeration. At one point I called the FindAllAsync with no parameters:

var myDevices = await Windows.Devices.Enumeration.DeviceInformation.FindAllAsync();

This did return a list of devices – which I could then find the Dream Cheeky Missile Launcher. However, when I tried to open the missile launcher the hidDevice was null.

# What Went Wrong?

After much searching and browsing it turns out that there is a reserved range of Usage Pages which defines HID devices. It turns out the Usage Page of 0x0001 defines a pointer. I recall that during the session on USB HID devices Microsoft mentioned that users would only be able to open non-reserved devices.

If you want to read up more on HID and usage pages start here:

<http://msdn.microsoft.com/en-us/library/windows/hardware/ff539946(v=vs.85).aspx>

As it turns out, the vendor would have to put their Usage Page starting in the range 0xFFxx to be usable.

So, where did that leave me?

I went out and bought this missile launcher:



It turns out that this device has the following information:

* Vidpid: 0x1941 0x8021
* UsagePage: 0xFFA0
* UsageID: 0x0001

The VID and PID are not critical; however, the UsagePage being 0xFFA0 will now work in Windows RT 8.1. This brings up an important point. If you have a vendor creating a custom USB device for you – or you are sourcing USB devices for a Windows Store application, you had better make sure that the UsagePage is defined correctly.

When I finally received the new launcher in the mail, I changed my capabilities to:

<Capabilities>

<m2:DeviceCapability Name="humaninterfacedevice">

<m2:Device Id="vidpid:1941 8021">

<m2:Function Type="usage:FFA0 0001" />

</m2:Device>

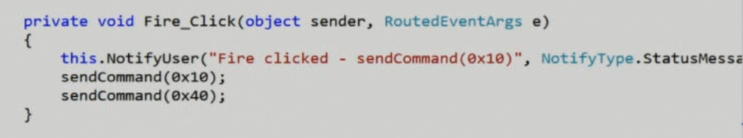
</m2:DeviceCapability>

</Capabilities>

I then tested the application again, and this time the USB device handle was returned properly. I was now finally ready to communicate with the device.

The WinRT 8.1 API makes it very easy to connect to a device. In just a few lines of code we have connected, and are ready to talk to the device.

If you watched the video presentation at build, you’ll know that everything I’ve written so far follows along with what you see in the video – right down to the part where all you can see is the API calls:



The presenter did not include the code for the sendCommand API in with the presentation.

It turns out – that connectivity is just part of the equation, the biggest part of the equation is communication. For this, you will require a programming specification for the USB device. The specification tells you what each API to the device is – along with the byte structure of the API. It will also explain when to send the bytes and when to read the bytes.

For the Dream Cheeky device it is controlled via 1 byte codes as follows:

|  |  |
| --- | --- |
| Up | 0x01 |
| Down | 0x02 |
| Left | 0x04 |
| Right | 0x08 |
| Fire | 0x10 |
| Stop | 0x20 |
| Status | 0x40 |

# Communicating with the USB HID Device

Communicating with a HID device is done via the OutputReport and InputReport. In this section I will show you how to a byte command to the launcher.

In our case we use a report Id of 0x00 and create an output report from our HID Device handle as follows:

ushort reportId = 0x00;

var outReport = hidDevice.CreateOutputReport(reportId);

If you have a look at the outReport you’ll notice that the device requires a byte buffer of 9 characters long. If we were using standard serial communications over USB – we wouldn’t need to package the total 9 byte message to the device. For our case, we only need to fill in the 1 byte as follows:

var dataWriter = new DataWriter();

dataWriter.WriteByte((Byte)reportId);

dataWriter.WriteByte((Byte)data);

The first byte is the report Id and the second byte is the launcher command. I did find a code sample that used the following code:

outReport.Data = dataWriter.DetachBuffer();

When I used the above code I found that the application would throw an exception "Value does not fall within the specified range". In this case I found that the dataWriter buffer was 99 bytes long, and the outReport.Data length was 9 bytes. Perhaps this will change in subsequent versions of WinRT 8.1, however, I found I could only set the outReport.Data to a 9 byte buffer.

To do this, I would grab a byte buffer from the out report, and populate it with the first 9 bytes of my command buffer:

//Create my command buffer

var dataWriter = new DataWriter();

dataWriter.WriteByte((Byte)reportId);

dataWriter.WriteByte((Byte)data);

//Create a buffer array from the command buffer

IBuffer newbuf = dataWriter.DetachBuffer();

byte[] newbufArray = WindowsRuntimeBufferExtensions.ToArray(newbuf);

//Create a buffer from the out report data – which will be 9 bytes. Copy it

byte[] array = WindowsRuntimeBufferExtensions.ToArray(outReport.Data);

for (int i = 0; i < array.Length && i<newbufArray.Length;i++)

{

array[i] = newbufArray[i];

}

IBuffer db2 = WindowsRuntimeBufferExtensions.AsBuffer(array);

//Set the out report data to the new command buffer.

outReport.Data = db2;

And now I send the data:

await hidDevice.SendOutputReportAsync(outReport);

The full method appears as:

public async void SendData(ushort data)

{

ushort reportId = 0x00;

var outReport = hidDevice.CreateOutputReport(reportId);

var dataWriter = new DataWriter();

dataWriter.WriteByte((Byte)reportId);

dataWriter.WriteByte((Byte)data);

IBuffer newbuf = dataWriter.DetachBuffer();

byte[] newbufArray = WindowsRuntimeBufferExtensions.ToArray(newbuf);

try

{

byte[] array = WindowsRuntimeBufferExtensions.ToArray(outReport.Data);

for (int i = 0; i < array.Length && i<newbufArray.Length;i++)

{

array[i] = newbufArray[i];

}

IBuffer db2 = WindowsRuntimeBufferExtensions.AsBuffer(array);

outReport.Data = db2;

System.Diagnostics.Debug.WriteLine("Sending:");

DebugBytes(new byte[] { (Byte)outReport.Id,(Byte)data });

try

{

await hidDevice.SendOutputReportAsync(outReport);

}

catch (Exception ex)

{

}

}

catch(Exception ex2)

{

}

return;

}