Austin Neidert and Cam Wrabel
April 17, 2017
USB HID Sample Application.pdf 1.0

USB HID Sample Application with Sample Code

The purpose of this application is to demonstrate how to communicate with a Microcom Model 485 printer using the USB HID interface. Specific communication features demonstrated here include communication preparation, sending information to the printer, and receiving information from the printer. This document assumes that the USB HID interface is already selected on the printer as the active communication interface being used. All sample code that appears in this document is included in the Microsoft Visual Studio 485HIDApp solution.

Step 1: Preparing Communication

Before data transfer between the host and the printer can occur, the communication endpoints must be set up. More specifically, before writing to the printer, a "write handle" is required to know where to write data. Likewise, before reading from the printer, a "read handle" is required to know where to read from.

The first step in preparing communication is to determine which of the currently available USB HID devices the desired device is. In order to do this the list of devices must be iterated over. An individual device in the list can be referenced using an integer `i` as an index passed into the `SetupDiEnumDeviceInterfaces` function. To retrieve more information about a specific device – such as the device path - the `SetupDiGetDeviceInterfaceDetail` function is used. After retrieving the device path, a handle is gotten using the `CreateFile` function. This handle can later be used as the write handle (used to write to the printer).

This handle can also be used to get the product name, manufacturer and serial number belonging to the device by calling `HidD_GetManufacturerString`, `HidD_GetProductString, and HidD_GetSerialNumberString` functions respectively. In this case, the manufacturer is "Microcom", the product is "485", and the serial number is a string up to 12 characters in length. After these properties are retrieved, `CreateFile` may be called again to get a second handle which will be used as the read handle (to read from the printer). In the sample application this all happens in a for loop within the `refreshPrinters` function. This function looks like the following:

```
public void refreshPrinters()
{
    hidDevicesDropDown.Items.Clear();
    Guid guid = new Guid();
    HidD_GetHidGuid(ref guid);
    IntPtr devInfoPtr = SetupDiGetClassDevs(ref guid, IntPtr.Zero, IntPtr.Zero, 0x12);
    int devInfoInt = (int)Marshal.PtrToStructure(devInfoPtr, typeof(int));
    securityAttributes.securityDescriptor = 0;
    securityAttributes.inheritHandle = System.Convert.ToInt32(true);
    securityAttributes.length = Marshal.SizeOf(securityAttributes);
    SP_DEVICE_INTERFACE_DATA deviceInterfaceData = new SP_DEVICE_INTERFACE_DATA();
    deviceInterfaceData.size = Marshal.SizeOf(deviceInterfaceData);
    for (uint i = 0; SetupDiEnumDeviceInterfaces(devInfoPtr, IntPtr.Zero, ref guid, i, ref deviceInterfaceData);
i++)
    {
```

```
        SP_DEVICE_INTERFACE_DETAIL_DATA spDeviceInterfaceDetailData = new SP_DEVICE_INTERFACE_DETAIL_DATA();
        uint requiredSize = 0;
        SetupDiGetDeviceInterfaceDetail(devInfoPtr, ref deviceInterfaceData, IntPtr.Zero, 0, ref requiredSize,
IntPtr.Zero);
        uint size = requiredSize;
        if (IntPtr.Size == 8) // for 64 bit operating systems
            spDeviceInterfaceDetailData.cbSize = 8;
        else
            spDeviceInterfaceDetailData.cbSize = 4 + Marshal.SystemDefaultCharSize; // for 32 bit systems
        IntPtr detailDataPtr = Marshal.AllocHGlobal(Marshal.SizeOf(spDeviceInterfaceDetailData));
        Marshal.StructureToPtr(spDeviceInterfaceDetailData, detailDataPtr, false);
        SetupDiGetDeviceInterfaceDetail(devInfoPtr, ref deviceInterfaceData, detailDataPtr, size, ref
requiredSize, IntPtr.Zero);
        spDeviceInterfaceDetailData = (SP_DEVICE_INTERFACE_DETAIL_DATA)Marshal.PtrToStructure(detailDataPtr,
typeof(SP_DEVICE_INTERFACE_DETAIL_DATA));
        string devicePath = spDeviceInterfaceDetailData.DevicePath;
        int handle = CreateFile(devicePath, 0xC0000000/*read and write*/, 0x3/*file share read and write*/, 0,
3/*open existing*/, 0, 0);//although this is specifying read and write this creates the first handle which will
be used for writing
        IntPtr buffer = Marshal.AllocHGlobal(100);
        HidD_GetSerialNumberString(handle, buffer, 100);
        string serialNumberString = Marshal.PtrToStringAuto(buffer);
        buffer = Marshal.AllocHGlobal(100);
        HidD_GetProductString(handle, buffer, 100);
        string productString = Marshal.PtrToStringAuto(buffer);
        buffer = Marshal.AllocHGlobal(100);
        HidD_GetManufacturerString(handle, buffer, 100);
        string manufacturerString = Marshal.PtrToStringAuto(buffer);
        int readHandle = CreateFile(devicePath, 0xC0000000/*read and write*/, 0x3/*file share read and write*/,
0, 3/*open existing*/, 0, 0);//although this is specifying read and write this creates the second handle which
will be used for reading
        Device d = new Device(devicePath, handle, readHandle, manufacturerString, productString,
serialNumberString);
        devices.Add(d);
        hidDevicesDropDown.Items.Add(d.manufacturer + " - " + d.product + " - " + d.serialNumber);
        if (devicePath.Contains(vid) && devicePath.Contains(pid))
            hidDevicesDropDown.SelectedIndex = (int)i;
    }
}
```

In the previous function all device paths, write handles, read handles, and feature strings are stored in the Device class. The Device class is defined in the sample application as a means to interact with the attached device. Once the correct device is known, the handles stored within that device object can be referenced.

Step 2: Writing to the Printer

Writing to the printer can be done any time after the write handle is retrieved. Writing to the handle is done using the `WriteFile` function. This can be done in-line with code as well as resulting from a button press.

```
public void sendBytes(byte[] outputBuffer)
{
    int written = 0;
    byte[] dataBuffer = new byte[33];
    int repCount = (outputBuffer.Length / (dataBuffer.Length - 1)); // number of reports to send is total data
size/report size
    if ((outputBuffer.Length % (dataBuffer.Length - 1) != 0)) // there is some "extra" data that doesn't fill a
whole report
        repCount++; //add another report and pad it with zeros
    for (int j = 0; j < repCount; j++) //loop through the reports to send
    {
        dataBuffer = new byte[33];
        int outputOffset = j * 32;
        for (int i = 1; i < dataBuffer.Length && outputOffset + i - 1 < outputBuffer.Length; i++) //fill dataBuf
with 32 values from outputBuffer then write
```

```
            dataBuffer[i] = outputBuffer[outputOffset + i - 1];
        WriteFile(currentDevice.handle, ref dataBuffer[0], dataBuffer.Length, ref written, 0);
    }
}
```

In the function above, all data that is sent must be split into 32 byte blocks. The required size of the buffer depends on the report descriptor of the device; for the Microcom 485 printer the report size is 32. When sending to a USB HID device the first byte in the buffer being sent is used as the "report number". In the case of the Model 485, the report number is unused. For this reason the starting value of i in the nested for loop is 1, leaving the first byte in the buffer as 0 (the 0th report). For many USB HID implementations, this report number would be incremented for each report to preserve the order of reports as they are received. In this case the Microcom 485 printer preserves order manually so the value of the report number is irrelevant. In summary, when sending data to a Microcom 485 printer, the buffer being sent should consist of an arbitrary one byte report number followed by 32 bytes of data.

Step 3: Reading from the Printer

Reading from a USB HID device shares many of the same properties as writing. Reading from the handle is done using the `ReadFile` function. Like sending data, this function uses a 33 byte buffer where the first byte is the report number, and the remaining 32 bytes are data. In the sample application, asynchronous I/O is used and thus all reading is done in an entirely new thread. Each call to `ReadFile` is done within a while loop to allow reading as soon as data is available. The entry to the read thread is shown below.

```
public void readThreadEntry()
{
    if (eventNum == 0)
        eventNum = CreateEvent(ref securityAttributes, 1, 1, "");
    HIDOverlapped.OffsetLow = 0;
    HIDOverlapped.OffsetHigh = 0;
    HIDOverlapped.EventHandle = (IntPtr)eventNum;
    int lastReadResult = 0;
    int readResult = 0;
    while (keepReading)
    {
        int read = 0;
        byte[] buffer = new byte[33];
        try
        {
            lastReadResult = readResult;
            ReadFile(currentDevice.readHandle, buffer, 33, ref read, ref HIDOverlapped);
            readResult = WaitForSingleObject(currentDevice.readHandle, 1000);
            if (readResult == 0)
            {
                string response = "";
                for (int i = 0; i < buffer.Length && i < read; i++)
                    response += (char)buffer[i];
                if (response.Length > 0)
                {
                    response = response.Replace(((char)0).ToString(), "");
                    threadSafeAddConsoleText(response);
                    if (response[response.Length - 1] == (char)0)
                        threadSafeAddConsoleText(Environment.NewLine);
                    uint count = (uint)HIDOverlapped.InternalHigh;
                    bool success = GetOverlappedResult(currentDevice.readHandle, ref HIDOverlapped, out count,
true);
                }
            }
```

```
            else if (readResult == 0x102)
                CancelIo(currentDevice.readHandle);
        }
        catch (ThreadAbortException tae) { }
        catch (Exception ex) { }
    }
    keepReading = true;//for the next read thread
}
```