HID Project: Electronic Music Instrument with Explorer 16 Development Board

BI3UI16: USB Integration

Georgina Challis
g.l.challis@student.reading.ac.uk
BEng Electronic Engineering

Jan 2018

Abstract *- A simple 'musical instrument' style peripheral was created by building on reference software for the Explorer 16 development board. Device firmware and a PC application were modified to communicate button presses and potentiometer states using commands over USB. The human inputs trigger sounds and visuals for music making. This basic prototype could easily be built upon further in many audio-based applications and improved to decrease input lag.*

Contents

Glossary

Term	Description
ADC	Analogue to Digital Converter
GUI	Graphical User Interface
HID	Human Interface Device
I/O	Input and Output
PIC	Programmable Integrated Circuit
PIM	Plug-In Module
USB	Universal Serial Bus

1. Introduction

The aim of this project was to create a new innovative product based on the Human Interface Device (HID) demo code available for the Explorer 16 development board. Inspired by the "reach out and touch" audio devices in Nicolas Collins' 2006 book '*Homemade Electronic Music: The Art of Hardware Hacking*'[1], the on-board buttons and potentiometer were used to create a USB controller that can be used to generate different sounds in a PC application.

Reference software from Microchip was used to implement a 2-way interface between the peripheral and a host PC, using sample code device firmware and a basic host application as a framework was built upon to create a new solution.

This report details the implementation of this project, including the results and potential improvements of the system. Code extracts and images of results can be found in the Appendix for further information.

2. Specifications

2.1. System Description

The product will be a hardware interface and supporting application to take user inputs and give varied audible and visual outputs. Development of the system will consist of 3 main parts: setting up the hardware, developing the HID firmware and building up the application for the host PC.

The hardware set-up will follow the recommended configuration for the Microchip Explorer 16 development board, as found on the product website [2] (see section 3).

The HID device will make use of the development board components to receive inputs from the user in real-time. These interactions will need to be communicated over USB to a host computer, which will display a graphical user interface (GUI) for the user to see and hear the results of their input. By pressing different buttons and turning the potentiometer to different positions, music can be played by combining the different audio outputs.

2.2. Requirements

In order to operate as a simple input device for audio, the peripheral and application must:

1. Communicate by sending messages over USB
2. React to button presses with visual indicators and audio outputs
3. React to the potentiometer being moved by changing between different audio 'modes'
4. Provide simple instructions to the user on the peripheral itself
5. Not fail when presented with multiple simultaneous button presses
6. Not crash when the peripheral is disconnected

Reasonable response speeds should be considered when testing the product to avoid a jarring user experience, but speeds matching real devices on the market are not expected for an initial attempt.

3. Existing Functionality

The Explorer board includes a number of different user interaction components. These include 4 push buttons and a potentiometer for user inputs and an LCD screen and LEDs for visual output [2]. The configuration used for this project includes a PIC32 Plug-In Module (PIM) which can be programmed with firmware to handle interaction and processing, and a PICtail Plus expansion board to communicate with a host computer via USB cable, as seen in Figure 1 below.

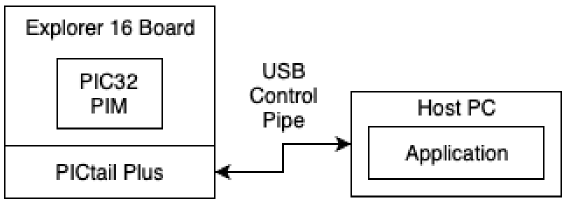


Figure 1. Hardware Configuration for Explorer 16 Board

To speed up the development time, demo software available from Microchip was used as a framework to build upon. This includes a sample Windows Forms application, written in C#, which displays the state of a single push-button, the value from a potentiometer position as a percentage bar and a status box that explains the current communication state. There is also a button to click to toggle the LEDs. The provided firmware for the peripheral device handles some of the inputs with a set of command codes to send and receive data across the USB control endpoint 0, enabling communication with the application [3].

4. Peripheral Firmware

4.1 Command Codes

Communication between the host and the device is simplified by taking advantage of the existing command codes in firmware. When an application needs the device to perform a certain operation, a hexadecimal code is sent over USB endpoint 0 to indicate which function is required. The HID regularly checks whether or not data was received from the host application, comparing the first element in the data buffer against defined command codes. If this matches a number in the switch statement, the corresponding block of code is executed. For example, if the device sees 0x80 in *ReceivedDataBuffer[0]*, the function to toggle the LEDs is performed. This project uses 0x81 to get the push-button states and 0x37 for reading the potentiometer.

4.1. Button Presses

The first part of this project was to detect button presses on all 4 of the Explorer board's push buttons. Each button was defined with its corresponding pins in the hardware profile header file '*HardwareProfile - PIC32MX460F512L PIM.h'* and set as switches 1 – 4 with pins RD6, RD7, RA7 and RD13 respectively. For example, switch 1 on the board is initialised at the start of the code with `mInitSwitch1()` where the digital bit D6 is set with `TRISDbits.TRISD6=1` and (see Appendix B for modifications to firmware code).

The code evaluates whether each switch has been pressed by checking the state of the defined pins (eg. `sw1` checks the state of `PORTDbits.RD6`). If the button is not pressed, the pull-up resistor on the board forces the PORT pin high (1). Once the button is pressed the pin goes low (0).

Once the pins were defined, all 4 buttons were added to command code 0x81, which originally returned a single button state. When command 0x81 is called, the first element of the data buffer is populated with the command code again to confirm the type of data being sent. The following 4 bytes of data contain the states of each button where 0x00 means the button is pressed down.

4.2. Potentiometer

When command 0x37 is called, the ReadPOT() method is called. The voltage at the potentiometer is measured and converted to a digital value with the Analogue to Digital Converter (ADC). As the pot is turned, the ADC values change to represent this with a value from 0 to 1024. Similar to the button command, the buffer is populated with the original hexadecimal command in byte 0 and then the following 2 bytes hold the most significant and least significant bytes for the 16-bit ADC value.

4.3. LCD Screen

The board's LCD screen was configured to meet the specification for providing simple user instructions. Pre-existing header files for configuring the LCD were imported into the project to set up the pins and functions for use. This made it simple to implement the LCDDisplayString() function and pass in strings of up to 16 characters for each of the two lines on the screen. This was used at the start of the while loop encompassing the main firmware code, so was initialised once on set-up and not modified throughout the use of the application.

5. Host Application

The HID demo code provides a rudimentary Graphical User Interface (GUI) created with Windows Forms to display the states of a single button, LED and potentiometer. This base C# code was used as a framework for the musical instrument application by leveraging other available .NET methods and writing a new Scales class for playing audio. A new GUI was built up with WinForms elements from the Visual Studio toolbox.

5.1. Requesting Peripheral States

In the ReadWrite thread of the application, packets containing the command codes are sent to the USB device with the WriteFile() function via the OUTBuffer. The first byte is the ReportID which is always 0 and is not sent over the USB bus. Following this is the command to return the ADC value or the push-button states. The remaining bytes of the packet are set to 1s to reduce power and stop toggling states on the line as per Non-Return to Zero Inverted (NRZI) encoding [7].

The results from the commands are then read from the INBuffer. The 2 bytes forming the ADC value are converted to the unsigned integer 'ADCValue' and the push button states are used to set Boolean states for 'PushButtonPressed' (a value of 0x00 sets PushButtonPressed to True 0x01 to False, because of the action of the pull-up resistor on the board). These variables are made global so they can be accessed throughout the application.

5.2. Playing Sounds

The audio outputs were created with 2 different methods. For the electronic 'piano' notes, the .NET Console method *Console.Beep()* was used. This takes two integers for the sound frequency and the duration in milliseconds and plays it through the PC's soundcard when called [4]. For drum sounds, 4 audio files were downloaded, cut down with the Audacity editor and embedded into the executable. When the drums were selected, each button corresponded to a sound file, playing using the *SoundPlayer* class in *System.Media* synchronously when the button is pressed [5].

It is possible to implement more complex libraries such as the NAudio toolkit for .NET to provide a wider range of realistic instrument sounds [6] but, with the limited number of buttons and potentiometer range, the project would not be able to take advantage of these extra options. This would also consume more memory for the program and increase the development time.

5.3. Notes and Scales Classes

With the development board having only 4 push buttons, it did not seem sufficient for the user to only be able to play 4 different notes in total. To resolve this, the potentiometer was used to switch between sets of notes in different 'scales'. The total range of the ADC was divided into 5 to give 5 distinct modes that the user could select by turning the pot (see Table 1 below).

Table 1: List of scales implemented in Scales.cs

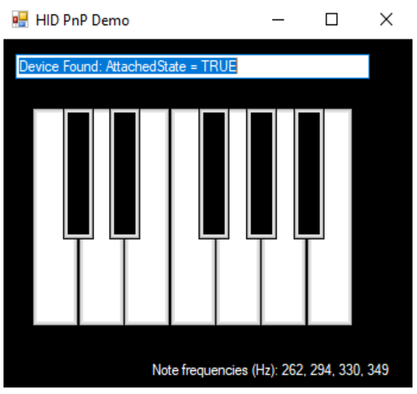
Scale Name	ADC Value	Note 1	Note 2	Note 3	Note 4
C Major (Part 1)	0 – 204	C1	D1	E1	F1
C Major (Part 2)	205 - 409	G1	A1	B1	C2
C Major Pentatonic (+1 Octave, - tonic)	410 – 614	D2	E2	G2	A2
B♭ Minor (Part 1)	615 – 819	A#*	C1	C#	D#
Drum Mode	820 - 1024	*Snap.wav*	*Kick.wav*	*Tom.wav*	*Snare.wav*

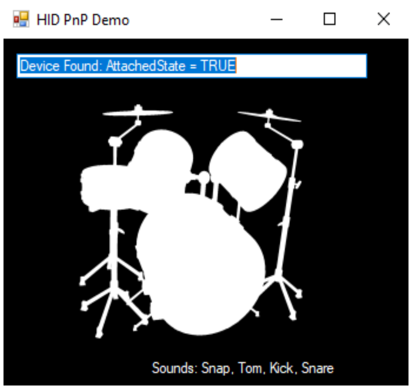
*Common Digital Audio Workstations (DAW), give notes with "sharps" names rather than "flats", i.e. A# = B♭

Values 0 – 409 contain the C Major scale between the first 2 modes to include all the white notes of a piano. To give some range, mode 3 increased the notes by an octave, doubling the frequencies. To keep this scale to 4 buttons, the pentatonic (5 note) scale was selected and the first C was removed. Mode 4 used the first part of the B♭ Minor scale to include some of the black piano notes, whilst not sounding dissonant when played together. The drum mode was selected when the ADC value was between 820 and 1024. This changes to use to Snap, Kick, Tom and Snare audio files for buttons 1 – 4 so does not make use of the Scales class.

5.4. Form Elements

The piano layout was created using 12 rectangular buttons from the WinForm toolbox, as shown in Figure 2(a) below. Each software button is linked to its corresponding note and, in addition to playing the relevant sound, changes the note's background colour when played with the physical push button.


HID PnP Demo
Device Found: AttachedState = TRUE
Note frequencies (Hz): 262, 294, 330, 349


HID PnP Demo
Device Found: AttachedState = TRUE
Sounds: Snap, Tom, Kick, Snare

(a) (b)

Figure 2. Final host application layout; (a) Piano keys layout; (b) Drum mode layout.

Note colours are handled with an if statement in the *Form.cs* code which checks whether the frequency is divisible by the note's first octave's frequency – a change of one octave equates to doubling the frequency. For example, if a note at 262Hz, 524Hz or 1,048kHz is played, the C piano key button will turn blue.

The label in the bottom right of the window shows the frequencies of notes in the selected mode and changes dynamically when the ADC value changes. For an ADC value greater than 820, the piano buttons become invisible and the drum image is displayed instead, as in Figure 2(b).

The status bar at the top of the window has been borrowed from the original demo GUI and shows the connectivity state of the Explorer board. If the device is plugged in and operating correctly, the bar should read "Device Found: AttachedState = TRUE". If the board cannot be found or is failing to communicate, this will read "Device Not Detected: Verify Connection/Correct Firmware".

6. Results

The results of functional testing can be seen in Figures 3 – 8 and Table 2 of Appendix A.

6.1. Communication via USB

The device was able to successfully communicate with the application. Figure 3 in shows the output of the status bar from the original Microchip GUI which confirms the device is connected.

6.2 Reacting to User Input

Figure 4 in shows the visual results of a single user input. When button 4 is pressed, the audio for an F note is played in the application and the correct key turns blue.

Figure 5 shows some of the GUI changes caused by turning the potentiometer. At 0, the starting mode lists frequencies of 262Hz, 294Hz, 330Hz and 349Hz (Fig. 5a). Around the mid-point, the label lists available frequencies of 588Hz, 660Hz, 784Hz and 880Hz (Fig. 5b). At the maximum potentiometer value, the background changes to the drum kit and the user can press the Snap, Toms, Kick Drum or Snare, as seen in Figure 5c.

6.3. Simple Instructions

Figure 6 shows the static state of the LCD screen. This gives a basic string of instructions to press buttons and turn the potentiometer to make sounds.

6.4. Simultaneous Buttons Presses

When multiple buttons are pressed at the same time, the GUI colours the background of all played notes, as in Figure 7. The Beep function is unable to play multiple notes at the same time however, so when the button states are polled, notes are played in sequence repetitively. Eg. If C, D and E are all pressed, the application will play C – D – E – C – D – E and so on until the combination changes.

6.5. Handle Disconnecting the Peripheral

Figure 8 shows the GUI state when the peripheral's USB connection or power is removed. The status bar tells the user "Device Not Detected: Verify Connection/Correct Firmware" and the label of available notes to play is no longer visible.

6.6. Response Time of Button Presses

Table 2 in Appendix A shows results of the time taken from pressing the button to the start of the audio output, recorded by stepping through each frame of a video recording of the output. The average response time for the audio was around 0.7s and 1.35s for the visuals.

7. Discussion

Overall, all the original specifications for the program were met. When a user presses the buttons, the application plays the appropriate sound and demonstrates the note visually. Turning the potentiometer through its full range appropriately selects different groups of sounds to play to give the user some variety.

The LCD screen successfully displays the static instructions text to tell the user what to do. This could do with some improvement by programming in a method to scroll characters across the screen over a set period of time, allowing for longer messages to be displayed. It would also be beneficial to implement a command code to tell the user which notes they have available on the screen when they switch modes to make it easier to use.

For responsiveness, the average time taken for audio to play is around 700ms. Compared to typical music controller latencies of around 0.5ms to 3ms [8], there is significant lag. For a small scale project, this is not too important and could potentially be improved by optimising the USB communication. For example, most USB musical interface devices tend to use bulk endpoints for sending data when bandwidth is available rather than polling, identifying as Audio class devices in their descriptor instead of HID devices [8] so this could be investigated further.

8. Further Development

This simple program could easily be expanded to form more complex solutions. With the existing functionality, the peripheral could use the LCD screen to give further information like dynamic updates of the selected scale and/or notes currently being played as previously mentioned. Having an extra 4 buttons on the device would also make the peripheral more useful for musical applications to represent a whole octave at a time.

Moving on from the development board, a custom-built device could use this configuration to enhance live music performances. A large array of push buttons could be used to trigger music loops for DJs in a similar way to the Ableton Push board [9], or the buttons could be replaced with movement sensors and used in wearable devices like the MI.MU Gloves for real-time music performance and audio manipulation [10].

The host application could also be made more sophisticated by implementing functions to record buttons presses over a set time, potentially saving and re-playing audio back to the user. Saving the notes to a text file, or even generating an output in musical notation could help aspiring musicians get started with composing and recording simple ideas without a full instrument or expensive software.

Additionally, the potential for both live performance and music production could be enhanced by integrating this product with other audio and visual software. By implementing the Musical Instrument Digital Interface (MIDI) standard, messages could be sent from the device to real-time graphics packages like MAX/MSP [11] and a Digital Audio Workstations (DAWs) such as Ableton Live, Pro Tools and Studio One to call up rhythmic visuals, record as an instrument or trigger pre-programmed macros instead of restrictive mouse clicking or inconvenient keyboards and drum pads.

9. Conclusion

Through this project, a simple human interface device was successfully used to create a music instrument style controller for a new application. Users can take advantage of existing interaction elements to play a variety of sounds from a connected host PC.

Making use of manufacturer reference software sped up the development time significantly, allowing the project to take advantage of the lower level USB communication and exception handling from demo code. This meant that time could be spent on building out the application functions and visuals. This simple framework could easily be taken further and built into a number of different music-based solutions.

References

[1] N. Collins, *Homemade Electronic Music: The Art of Hardware Hacking,* 2nd ed. New York: Routledge, 2006.

[2] Microchip (2014). *Explorer 16 Development Board User's Guide* [Online] Available: http://ww1.microchip.com/downloads/en/devicedoc/50001589b.pdf

[3] Microchip (2018). *USB Endpoints* [Online] Available: http://microchipdeveloper.com/usb:endpoints

[4] Microsoft. *.NET API Browser | Console.Beep Method* [Online] Available: https://docs.microsoft.com/en-us/dotnet/api/system.console.beep?view=netframework-4.7.2

[5] Microsoft. *.NET API Browser | SoundPlayer Class* [Online] Available: https://docs.microsoft.com/en-us/dotnet/api/system.media.soundplayer?view=netframework-4.7.2

[6] M. Heath (2008, Jun) *Introducing NAudio - .NET Audio Toolkit* [Online] Available: https://markheath.net/post/introducing-naudio-net-audio-toolkit

[7] J. Slobodov *Universal Serial Bus (USB)* [Online] Available: http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2001f/interfacing/usb/appl_note.html

[8] B. G. Schultz (2018). *The Schultz MIDI Benchmarking Toolbox for MIDI interfaces, percussion pads and sound cards* [Online] Available: https://link.springer.com/content/pdf/10.3758%2Fs13428-018-1042-7.pdf

[9] Ableton (2018). *Push: Music at your Fingertips* [Online] Available: https://www.ableton.com/en/push/

[10] MI.MU Gloves Ltd. (2018). *MI.MU: Tech* [Online] Available: https://mimugloves.com/tech/

[11] J. Hass, Indiana University (2017) *How does the MIDI system work?* [Online] Available: http://www.indiana.edu/~emusic/etext/MIDI/chapter3_MIDI13.shtml

All web addresses referred to in this report were verified on 30th January 2019.

Appendices

Appendix A – Results Images

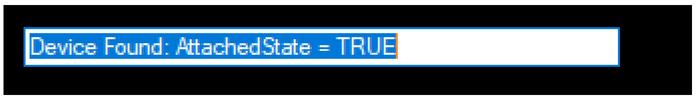
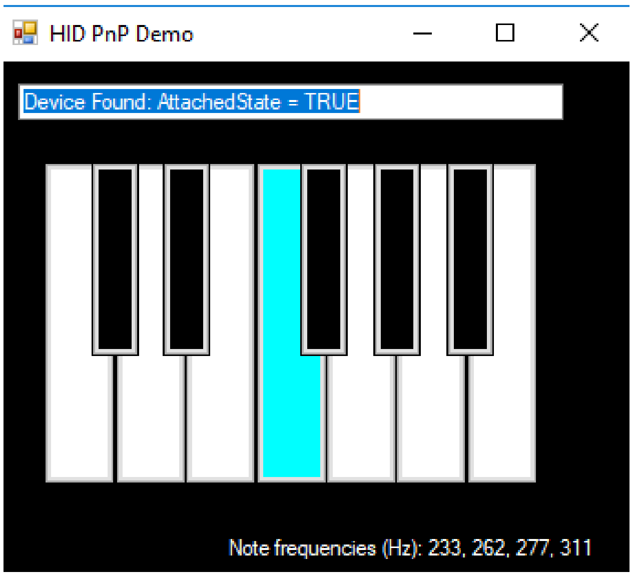

Device Found: AttachedState = TRUE

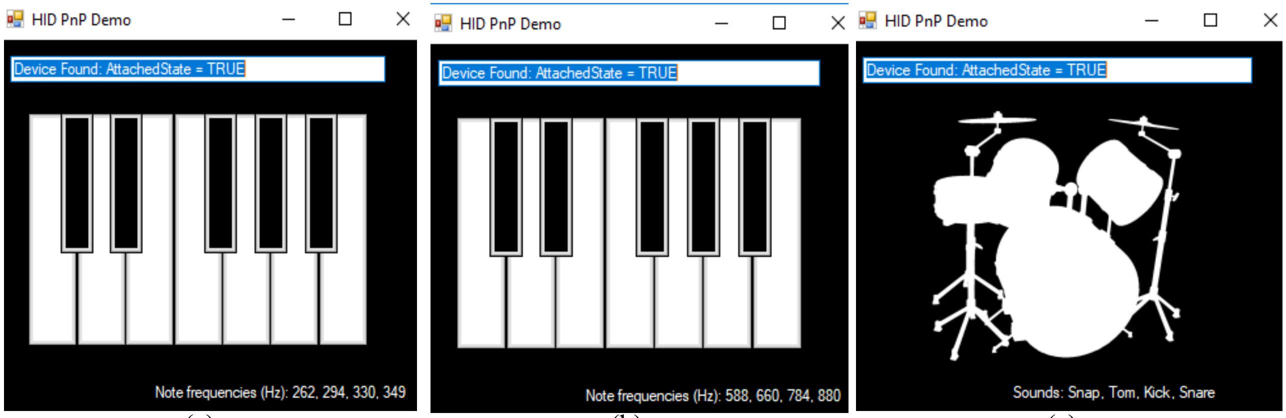
Figure 3. Status bar confirming USB connection



Device Found: AttachedState = TRUE

Note frequencies (Hz): 233, 262, 277, 311

Figure 4. Single button press response.



Device Found: AttachedState = TRUE

Note frequencies (Hz): 262, 294, 330, 349

Device Found: AttachedState = TRUE

Note frequencies (Hz): 588, 660, 784, 880

Device Found: AttachedState = TRUE

Sounds: Snap, Tom, Kick, Snare

(a) (b) (c)

Figure 5. Comparison of different modes; (a) C Minor Part 1; (b) C Pentatonic (- tonic); (c) Drums

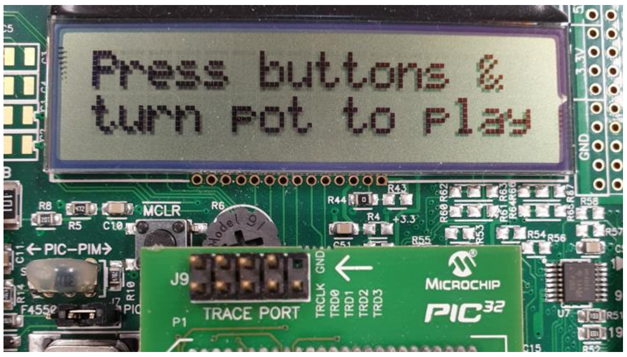
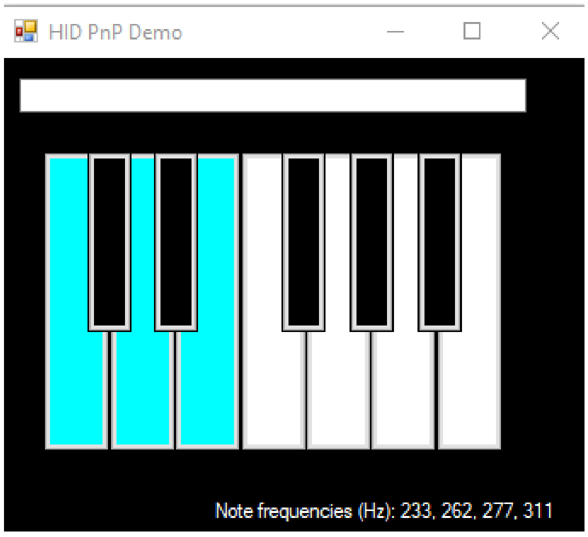
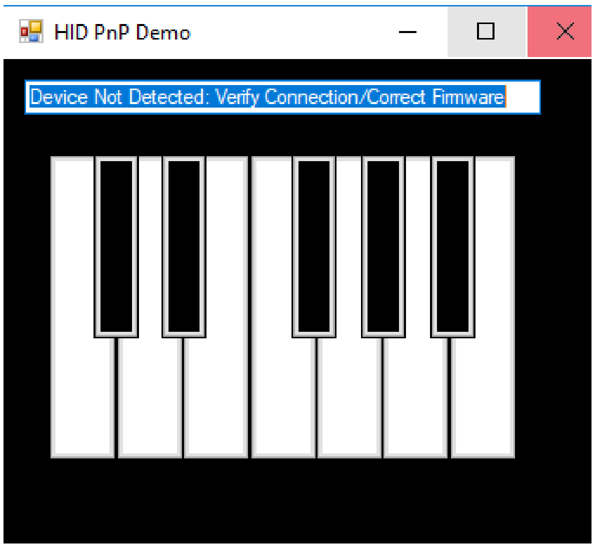

Press buttons &
turn pot to play

Figure 6. LCD screen showing basic instructions



Note frequencies (Hz): 233, 262, 277, 311

Figure 7. Visual response to multiple simultaneous inputs



Device Not Detected: Verify Connection/Correct Firmware

Figure 8. GUI when peripheral is disconnected

Table 2. Time of audible response from button presses

Button Pressed	Time Delay – Visuals (secs)	Time Delay – Audio (secs)
RD6	1.2	0.5
RD7	1.2	0.5
RA7	1.8	0.8
RD13	1.2	1
Average	1.35	0.7

Appendix B – MPLAB Firmware Code Extracts

#define HARDWARE_PROFILE_PIC32MX460F512L_PIM_H

```
//...
   /** SWITCH *********************************************************/
   #define mInitSwitch1()      TRISDbits.TRISD6=1;
   #define mInitSwitch2()      TRISDbits.TRISD7=1;
   #define mInitSwitch3()      TRISAbits.TRISA7=1;
   #define mInitSwitch4()      TRISDbits.TRISD13=1;

   #define mInitAllSwitches()  mInitSwitch1();mInitSwitch2();mInitSwitch3();mInitSwitch4();

   #define sw1              PORTDbits.RD6
   #define sw2              PORTDbits.RD7
   #define sw3              PORTAbits.RA7
   #define sw4              PORTDbits.RD13

   /** POT ************************************************************/
   #define mInitPOT()  {AD1PCFGbits.PCFG5 = 0;    AD1CON2bits.VCFG = 0x0;    AD1CON3bits.ADCS = 0xFF;    AD1CON1bits.SS
RC = 0x0;    AD1CON3bits.SAMC = 0x10;    AD1CON1bits.FORM = 0x0;    AD1CON2bits.SMPI = 0x0;    AD1CON1bits.ADON = 1;}

//...
```

#define MAIN_C

```
//...
#include "./USB/usb.h"
#include "HardwareProfile.h"
#include "./USB/usb_function_hid.h"

#include "lcd_demo.h"
#define LCD_LINE_ONE
#define LCD_LINE_TWO

unsigned char ReceivedDataBuffer[64];
unsigned char ToSendDataBuffer[64];
```

int main(void)

```
{
   InitializeSystem();

      PMMODE = 0x03ff;
      PMCON = 0x8383;
      PMAEN = 0x0001;
      LCDInit();

   #if defined(USB_INTERRUPT)
      USBDeviceAttach();
   #endif

   LCDClear();       //to clear the LCD
   LCDL1Home();    //to go to the start of line 1
   LCDDisplayString((BYTE*)"Press buttons &", LCD_LINE_ONE);//display the string to line1
   LCDDisplayString((BYTE*)"turn pot to play", LCD_LINE_TWO); //display the string to line2

   while(1)
   {
      // Check bus status and service USB interrupts.
      USBDeviceTasks();
      ProcessIO();
   }//end while
}//end main
```

void ProcessIO(void)

```
{
   // User Application USB tasks
   if((USBDeviceState < CONFIGURED_STATE)||(USBSuspendControl==1)) return;

   if(!HIDRxHandleBusy(USBOutHandle))  //Check if data was received from the host.
   {
    switch(ReceivedDataBuffer[0])//Look at the data the host sent, to see what kind of application specific command it sent.
      {
            case 0x81:  //Get push button state
               ToSendDataBuffer[0] = 0x81;     //Echo back to the host PC the command
               if(sw1 == 1)                    //pushbutton not pressed
               { ToSendDataBuffer[1] = 0x01;}
               else                            //sw3 must be == 0, pushbutton is pressed
               { ToSendDataBuffer[1] = 0x00;}
               if(sw2 == 1)
               { ToSendDataBuffer[2] = 0x01;}
               else
               { ToSendDataBuffer[2] = 0x00;}
               if(sw3 == 1)
               { ToSendDataBuffer[3] = 0x01;}
               else
               { ToSendDataBuffer[3] = 0x00;}
               if(sw4 == 1)
               { ToSendDataBuffer[4] = 0x01;}
               else
               { ToSendDataBuffer[4] = 0x00;}

               if(!HIDTxHandleBusy(USBInHandle))
               { USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64); }
               break;

            case 0x37:    //Read POT command.
               {
                  WORD_VAL w;

                  if(!HIDTxHandleBusy(USBInHandle))
                  {
                     mInitPOT();
                     w = ReadPOT();    //Use ADC to read the I/O pin voltage.
                     ToSendDataBuffer[0] = 0x37;      //Echo back to the host the command
                     ToSendDataBuffer[1] = w.v[0];      //Measured analog voltage LSB
                     ToSendDataBuffer[2] = w.v[1];      //Measured analog voltage MSB
                     USBInHandle = HIDTxPacket(HID_EP,(BYTE*)&ToSendDataBuffer[0],64);
                  }
               }
               break;
         }
         //Re-arm the OUT endpoint for the next packet
         USBOutHandle = HIDRxPacket(HID_EP,(BYTE*)&ReceivedDataBuffer,64);
   }
}//end ProcessIO
```

```
WORD_VAL ReadPOT(void)
{
   WORD_VAL w;
   // Routine to read the Explorer 16 potentiometer.
            // Get an ADC sample
         AD1CHS0bits.CH0SA = 5;\
         AD1CON1bits.SAMP = 1;           //Start sampling
         for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
         AD1CON1bits.SAMP = 0;           //Start sampling
         for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
         while(!AD1CON1bits.DONE);       //Wait for conversion to complete

      w.Val = ADC1BUF0;

   return w;
}//end ReadPOT

//...
```

Appendix C – WinForms Main Form Extracts

```
//Variables used by the application/form updates.
//Updated by ReadWriteThread,read by FormUpdateTimer tick handler (needs to be atomic)
bool PushbuttonPressed = false;
bool Pushbutton2Pressed = false;
bool Pushbutton3Pressed = false;
bool Pushbutton4Pressed = false;
uint ADCValue = 0;
SoundPlayer soundplayer;


private void ReadWriteThread_DoWork(object sender, DoWorkEventArgs e){
                Byte[] OUTBuffer = new byte[65];    //Allocate a memory buffer equal to the OUT endpoint size + 1
                Byte[] INBuffer = new byte[65];     //Allocate a memory buffer equal to the IN end         point size + 1
                uint BytesWritten = 0;
                uint BytesRead = 0;

                while (true)
        {
        try
         {
        if (AttachedState == true) //Do not try to use the read/write handles unless the USB device is attached and ready
           {
           OUTBuffer[0] = 0x00;    //The first byte is the "Report ID" and does not get sent over the USB bus.  Always set = 0.
           OUTBuffer[1] = 0x37;    //READ_POT command (see the firmware source code), gets 10-bit ADC Value
                     for (uint i = 2; i < 65; i++).  OUTBuffer[i] = 0xFF;
//To get the ADCValue, first, we send a packet with our "READ_POT" command in it.
                if (WriteFile(WriteHandleToUSBDevice, OUTBuffer, 65, ref BytesWritten, IntPtr.Zero))    //Blocking function, unless an "over
lapped" structure is used
                 {
                  INBuffer[0] = 0;
//Now get the response packet from the firmware.
                  if (ReadFileManagedBuffer(ReadHandleToUSBDevice, INBuffer, 65, ref BytesRead, IntPtr.Zero))  //Blocking function, unles
s an "overlapped" structure is used
                   {
                    if (INBuffer[1] == 0x37)
                    {
                     ADCValue = (uint)(INBuffer[3] << 8) + INBuffer[2];//Need to reformat the data from two unsigned chars into one unsigne
d int.
                     }
                   }
                 }
//Get the pushbutton state from the microcontroller firmware.
                 OUTBuffer[0] = 0;
                 OUTBuffer[1] = 0x81; //0x81 is the "Get Pushbutton State" command
                 for (uint i = 2; i < 65; i++)//This loop is not strictly necessary.
                     OUTBuffer[i] = 0xFF;
//To get the pushbutton state, first, we send a packet with our "Get Pushbutton State" command in it.
        if (WriteFile(WriteHandleToUSBDevice, OUTBuffer, 65, ref BytesWritten, IntPtr.Zero))    //Blocking function, unless an "overlappe
d" structure is used
        {
        //Now get the response packet from the firmware.
        INBuffer[0] = 0;
          {
          if (ReadFileManagedBuffer(ReadHandleToUSBDevice, INBuffer, 65, ref BytesRead, IntPtr.Zero))    //Blocking function, unless an
"overlapped" structure is used
            {
            if (INBuffer[1] == 0x81) {
             if (INBuffer[2] == 0x01) PushbuttonPressed = false;
             else if (INBuffer[2] == 0x00) PushbuttonPressed = true;

             if (INBuffer[3] == 0x01) Pushbutton2Pressed = false;
             else if (INBuffer[3] == 0x00) Pushbutton2Pressed = true;

             if (INBuffer[4] == 0x01) Pushbutton3Pressed = false;
             else if (INBuffer[4] == 0x00) Pushbutton3Pressed = true;

             if (INBuffer[5] == 0x01) Pushbutton4Pressed = false;
             else if (INBuffer[5] == 0x00) Pushbutton4Pressed = true;
            }
          }
        }
    }
}
```

```
else {
   Thread.Sleep(5); //Add a small delay
 }
}
catch {
//Exceptions can occur during the read or write operations. For example,
exceptions may occur if for instance the USB device is physically unplugged
from the host while the above read/write functions are executing.

//Don't need to do anything special in this case.  The application will automatically
re-establish communications based on the global AttachedState boolean variable used
in conjunction with the WM_DEVICECHANGE messages to dyanmically respond to Plug and Play
USB connection events.
    }
 }
}

private void FormUpdateTimer_Tick(object sender, EventArgs e)
{

//...

//Update the various status indicators on the form with the latest info obtained from the ReadWriteThread()
  if (AttachedState == true)
  {
    if (ADCValue < 820)
    {
      MusicScale chosenScale = new MusicScale((int)ADCValue);
      Note chosenNote;
      ANxVoltage_lbl.Text = "Note frequencies (Hz): " + chosenScale.Notes[0].tone.ToString() + ", " + chosenScale.Notes[1].tone.ToStrin
g() + ", " + chosenScale.Notes[2].tone.ToString() + ", " + chosenScale.Notes[3].tone.ToString();


    foreach (var b in this.Controls.OfType<Button>()) { b.Visible = true; }

    pictureBox1.Visible = false;

    Console.WriteLine(PushbuttonPressed + "\t" + Pushbutton2Pressed + "\t" + Pushbutton3Pressed + "\t" + Pushbutton4Pressed);
    //Update the pushbutton state label.
    if (PushbuttonPressed == false)
    {
      button_C.BackColor = Color.White;
    }
    else
    {
     button_C.BackColor = Color.Cyan;
     chosenNote = chosenScale.Notes[0]; }
     Console.Beep(chosenNote.tone, chosenNote.duration); //Note C, quarter note
    }
    if (Pushbutton2Pressed == false)
    {
      button_D.BackColor = Color.White;
    }
    else
    {
      button_D.BackColor = Color.Cyan;
      chosenNote = chosenScale.Notes[1];
      Console.Beep(chosenNote.tone, chosenNote.duration); //Note C, quarter note
    }
    if (Pushbutton3Pressed == false)
    {
      button_E.BackColor = Color.White;
    }
    else
      {
       button_E.BackColor = Color.Cyan;
       chosenNote = chosenScale.Notes[2];
       Console.Beep(chosenNote.tone, chosenNote.duration); //Note C, quarter note
      }
      if (Pushbutton4Pressed == false)
      {
       button_F.BackColor = Color.White;
      }
      else
      {
            button_F.BackColor = Color.Cyan;
            chosenNote = chosenScale.Notes[3];
            Console.Beep(chosenNote.tone, chosenNote.duration); //Note C, quarter note
           }
         }
     else {
       foreach (var b in this.Controls.OfType<Button>()) { b.Visible = true; }
       pictureBox1.Visible = true;

       ANxVoltage_lbl.Text = "Sounds: Snap, Tom, Kick, Snare";
       string audioFile ="NULL";
       if (PushbuttonPressed) { audioFile = Path.GetFullPath("Properties/6.wav"); }
       else if (Pushbutton2Pressed) { audioFile = Path.GetFullPath("Properties/2.wav"); }
       else if (Pushbutton3Pressed) { audioFile = Path.GetFullPath("Properties/3.wav"); }
       else if (Pushbutton4Pressed) { audioFile = Path.GetFullPath("Properties/7.wav"); }
       else return;

       soundplayer = new SoundPlayer(audioFile);
       soundplayer.PlaySync();
      }
     Thread.Sleep(30);
    }

//...

}
```

Appendix D – Scales.cs Class

```
using System;

// Tone enum contains freq. of all notes used within the program
// (1 octave of black notes, 3 octaves of white notes)
public enum Tone
{
    REST = 0,
    As = 233, Cs = 277, Ds = 311, Fs = 370, Gs = 415,
    C1 = 262,   D1 = 294,   E1 = 330,   F1 = 349, G1 = 392,   A1 = 440, B1 = 247*2,
    C2 = C1*2,  D2 = D1*2,  E2 = E1*2,  G2 = G1*2,  A2 = A1*2,
    C3 = C1*4,  D3 = D1*4,  E3 = E1*4,  G3 = G1*4,  A3 = A1*4,
    C4 = C1*8
}

//Duration of note to be played (in milliseconds)
public enum Duration
{
    WHOLE = 1600,
    HALF = WHOLE / 2,
    QUARTER = HALF / 2,
    EIGHTH = QUARTER / 2,
    SIXTEENTH = EIGHTH / 2,
}

//Class for a single note-----------------------
public class Note {

    public int tone;
    public int duration;

    public Note(int frequency, int time)
    {
        tone = frequency;
        duration = time;
    }
}

//Class for the whole list of Scales ------------
public class MusicScale {
    public Note[] Notes;
    public MusicScale(int potVal)
    {
        if (potVal < 205)
        {
            Notes = new Note[]{
            new Note((int) Tone.C1, (int) Duration.QUARTER),
            new Note((int) Tone.D1, (int) Duration.QUARTER),
            new Note((int) Tone.E1, (int) Duration.QUARTER),
            new Note((int) Tone.F1, (int) Duration.QUARTER),
            };
        }
        else if (potVal < 410)
        {
            Notes = new Note[]{
            new Note((int) Tone.G1, (int) Duration.QUARTER),
            new Note((int) Tone.A1, (int) Duration.QUARTER),
            new Note((int) Tone.B1, (int) Duration.QUARTER),
            new Note((int) Tone.C2, (int) Duration.QUARTER),
            };
        }
        else if (potVal < 615)
        {
            Notes = new Note[]{
                new Note((int) Tone.D2, (int) Duration.QUARTER),
                new Note((int) Tone.E2, (int) Duration.QUARTER),
                new Note((int) Tone.G2, (int) Duration.QUARTER),
                new Note((int) Tone.A2, (int) Duration.QUARTER)
            };
        }
        else if (potVal < 820)
        {
            Notes = new Note[]{
            new Note((int) Tone.As, (int) Duration.QUARTER),
            new Note((int) Tone.C1, (int) Duration.QUARTER),
            new Note((int) Tone.Cs, (int) Duration.QUARTER),
            new Note((int) Tone.Ds, (int) Duration.QUARTER),
            };
        }

        else {
            Notes = new Note[]{
            new Note((int) Tone.C1, (int) Duration.QUARTER),
            new Note((int) Tone.D1, (int) Duration.QUARTER),
            new Note((int) Tone.E1, (int) Duration.QUARTER),
            new Note((int) Tone.G1, (int) Duration.QUARTER),
            };
        }
    }
}
```