BY FRED EADY

GO NUTS WITH THE KADTRONIX USB HID API LIBRARY

As a kid, I knew them as "hicker nuts." What my little Southern mouth was trying to say was hickory nuts. If you're a *Nuts & Volts* reader living in the southeastern United States, at one time or another you have probably been out there rooting around with the squirrels collecting hickory nuts.

For those of you that don't know what a hickory nut is, it is a cousin to the pecan and grows wild in my hometown of Fayetteville, TN. A typical Tennessee hicker nut is a bit smaller than a quarter and is contained within a very hard outer shell. As a kid, my mother and I would go into the woods on my grandma's farm in search of hicker nuts armed only with a couple of claw hammers. Hickory nuts have to be shelled with a hammer and a rock as the casings are too hard to crack with your hands or teeth. We would eat the good ones we found on the spot, then move on to the next prolific nut location once we had rooted out all of the good nuts. The tell-tale sign of a bad hicker nut is a small hole drilled into the shell by a worm or other strong-mouthed insect. Our foraging would last for hours since hickory nut meat is very small, and every hammer smack didn't guarantee that an edible nut was waiting inside.

Hunting hicker nuts is very similar to wading through Microsoft Visual C++ code. There's lots of hard outer shell and cracking through doesn't always mean you'll find what you're looking for. You also have to know where the trees are. If you can't find the trees, you can't pick up nuts. You can't crack the nuts if you can't find them. If you do manage to find some nuts, you'd better have the correct tools to crack them.

THE KADTRONIX USB HID API LIBRARY

The Kadtronix USB HID API Library was initially designed to support Visual Basic 6 and Visual C++ 6. Later, the Kadtronix folks wrote a version that supports the .NET version of Visual C++. Although you can still purchase Visual Basic 6 and Visual C++ 6 from vendors on the Internet, at the time of this writing the latest and greatest versions of Visual Basic and Visual C++ could only be had in Microsoft's Visual Studio 2008 product. (Visual Studio 2010 is supposedly due to be released in March.) So, rather than base this month's discussion on the legacy Visual Basic and Visual C++ compilers, I decided to show you how to move data between a PC and our PIC32MX795F512L TRAINER with the compilers contained within the Microsoft Visual Studio 2008 Standard. This is possible due to the Kadtronix Library's affinity for compilers that can access dynamic link libraries (DLLs).

After a couple of days attempting to use the Visual Studio version of Visual Basic with the Library, I found that it and the .NET-based Visual Basic component of Visual Studio 2008 are incompatible as they stand. Most of my grief was created by the differences in the ways Visual Basic 6 and the .NET Visual Basic component handle string data. The API Library uses Visual Basic 6 string handling methods that just don't exist in the .NET component of Visual Studio 2008. The Kadtronix documentation points out that the API Library may not support the .NET Framework in the Visual Basic environment. I felt that I owed it to the Design Cycle readers to put the Visual Studio version of Visual Basic to the test myself. The Kadtronix folks were right on the money.

The Kadtronix Library package includes a .NET DLL (UsbHidApi_NET.dll) for Visual C++. However, I was able to combine the Visual Studio 2008 version of Visual C++ and the standard Visual C++ Kadtronix Library DLL (UsbHidApi.dll) to produce the HID data transfer code that we will be discussing.

The UsbHidApi.lib file (which is included with the Library package) is also a necessary part of the Visual C++ compilation and link process. I explicitly defined the UsbHidApi.lib to the Visual C++ linker from within the VS 2008 configuration controls. The UsbHidApi.dll file is embedded within the Windows/system32 directory during

the installation of the Library package. Okay, now that you know where the trees are, let's crack some nuts.

TOPSEY TURVEY

Normally, we would discuss the microcontroller side of things before adding the PC ingredients into the mix. Instead, let's start by sorting through the Microsoft hicker nuts. The first nut we come across is the Kadtronix UsbHidApi.lib file which contains the interface functions and definitions that reach into the bowels of the Windows HID service engine. This file allows us to easily manipulate Windows' built-in HID features without an in-depth knowledge of Windows OS programming. As I mentioned earlier, the UsbHidApi.lib file is supported by a DLL which contains other important data items and functions that will shield us from the heat of raw Windows programming.

We've all heard the stories about desperate firmware thieves slithering off the packages of code-protected microcontrollers and stealing their code by examining the exposed silicon with a microscope. Once they had a "photograph" of the microcontroller's silicon layout, they could reverse-engineer the protected functions. A like situation exists with a .lib or .dll file. If we just absolutely had to know how the library or DLL works, we could use some expensive software tools to break it down and analyze the assembler code. But who really wants to do that? After all, obtaining the tools means nothing if you don't have the skills to interpret and rewrite the code. If the would-be firmware cheat had the knowledge to reverse-engineer the stolen assembler mnemonics, he or she wouldn't need to steal the code because they could write the necessary algorithms from scratch.

Unlike that code-protected microcontroller, we have a road map that helps us navigate the .lib and .dll functionality of the Kadtronix Library. That road map is in the guise of a file called UsbHidApi.h which is part of the Kadtronix install package.

The UsbHidApi.h file contains the information we need to be able to access the internal functionality of the UsbHidApi.lib and UsbHidApi.dll library files. For instance, here are the Read and Write HID library function declarations found within UsbHidApi.h:

```
extern "C" int _stdcall Read(void *pBuf);
    // Read from the HID device
extern "C" int _stdcall Write(void *pBuf);
    // Write to the HID device
```

The Library also houses other types of data declarations in the UsbHidApi.lib and UsbHidApi.dll files which include character and integer variable definitions, structures, and inheritable class instances.

Although we will only be addressing a single HID-class device, the API Library has the ability to query and identify multiple HID-class devices. Each device's parameters are stored in a structure called mdeviceList which is revealed to us in UsbHidApi.h. Here's what the mdeviceList structure looks like:

```
typedef struct {
  char      DeviceName[50];
    // Device name
  char      Manufacturer[50];
    // Manufacturer
  char      SerialNumber[20];
    // Serial number
  unsigned int VendorID;
    // Vendor ID
  unsigned int ProductID;
    // Product ID
  int       InputReportLen;
    // Length of HID input report (bytes)
  int       OutputReportLen;
    // Length of HID output report (bytes)
  int       Interface;
    // Interface
  int       Collection;
    // Collection
 } mdeviceList;
```

Instead of me running my mouth about the mdeviceList structure, let's write some C code to fill its variables with information we obtain from the PIC32MX795F512L TRAINER.

THE GETLIST API CALL

If multiple HID-class devices need to be uniquely serviced by our HID host, we must be able to identify each of them individually. That's where the GetList API call comes in:

```
int GetList(unsigned int VendorID,
              // Vendor ID to search
              // (0xffff if unused)
            unsigned int ProductID,
              // Product ID to search
              // (0xffff if unused)
            char          *Manufacturer,
              // Manufacturer (NULL if unused)
            char          *SerialNum,
              // Serial number to search
              // (NULL if unused)
            char          *DeviceName,
              // Device name to search
              // (NULL if unused)
            mdeviceList  *pList,
              // Caller's array for storing
              // matching device(s)
            int           nMaxDevices);
              // Size of the caller's array list
              // (no.entries)
```

The GetList function is part of the class imported from the UsbHidApi DLL called CUsbHidApi. If an attached HID-class device is available, invoking GetList will pull its operating parameters into a slot of the mdeviceList structure array. Before we can invoke GetList, we need to bring an instance of CUsbHidApi called hidDevices to life. The name hidDevices is arbitrary:

```
CUsbHidApi    hidDevices;
```

Using the mdeviceList structure outlined in the UsbHidApi.h file as a template, we must also create an array of structures which we will call m_DeviceList. Note that there are only two device structures available in our m_DeviceList[2] array. If we needed to track more than two devices, we would simply allocate the necessary

number of structure array entries. Right now, we are limited to detecting only two devices. In reality, we only require a single structure in our array. Our GetList call will reside inside of the DetectDevice function. So, we'll also code the DetectDevice function prototype into dcHidApidlg.h:

```
//declared public in dcHidApiDlg.h
mdeviceList m_DeviceList[2];
void DetectDevice(void);
```

The next step involves declaring the variables that the GetList function requires. The variable declaration process includes initialization of *pList which is loaded to point to the array of structures we created in dcHidApiDlg.h. Every variable declared can be traced back to its roots in the GetList function that was imported from the UsbHidApi.dll:

```
//this code is part of dcHidApiDlg.cpp
void CdcHidApiDlg::DetectDevice(void)
{
        unsigned int activeDevices,
        vendor_id,product_id;
        char  *manufacturer,*serial_num,
        *device_name;
        mdeviceList *pList = m_DeviceList;
```

Once we have declared all of the variables we need for the GetList function, we prepare the GetList function to get information on every HID-class device that is available by not specifying any information in the call that can be traced to any particular HID-class device that has made itself available:

```
vendor_id = 0xFFFF;
        // 0xFFFF => Any vendor ID
product_id = 0xFFFF;
        // 0xFFFF =>Any product ID
serial_num = NULL;
        // NULL=> Any serial number
device_name = NULL;
        // NULL=> Any device name
manufacturer = NULL,
        // NULL=> Any manufacturer
```

Upon invocation, the GetList API function will return the number of HID-class devices that respond to the function's request. In addition, each device that responds will fill a structure entry in the array of structures in m_DeviceList which is referenced by the pointer *pList:

```
activeDevices = hidDevices.GetList(
   vendor_id,           // Vendor ID
   product_id,          // Product ID
   manufacturer,        // Manufacturer
   serial_num,          // Serial number
   device_name,         // Device name
   pList,               // Device list
   2);                  // max number of
                        // devices
```

The data that flows from a responding device to the m_DeviceList structure originates in the TRAINER's device descriptor.

DESCRIPTOR REVIEW

Before we execute the GetList API function, let's grease the skids and take a look at what to expect in the TRAINER's m_DeviceList structure entry. As I mentioned earlier, every piece of data returned to the m_DeviceList structure will be gleaned from a PIC TRAINER descriptor entry. The Microchip MCHPFSUSB Framework takes the hard work out of descriptor creation.

The first element of our m_DeviceList structure is the DeviceName which is a 50 character array. I entered the following device name into the TRAINER descriptor file:

```
//Product string descriptor
ROM struct{BYTE bLength;BYTE bDscType;WORD
string[12];}sd002={
sizeof(sd002),USB_DESCRIPTOR_STRING,
{'O','U','R','H','I','D','D','E','V','I','C','E'
}};
```

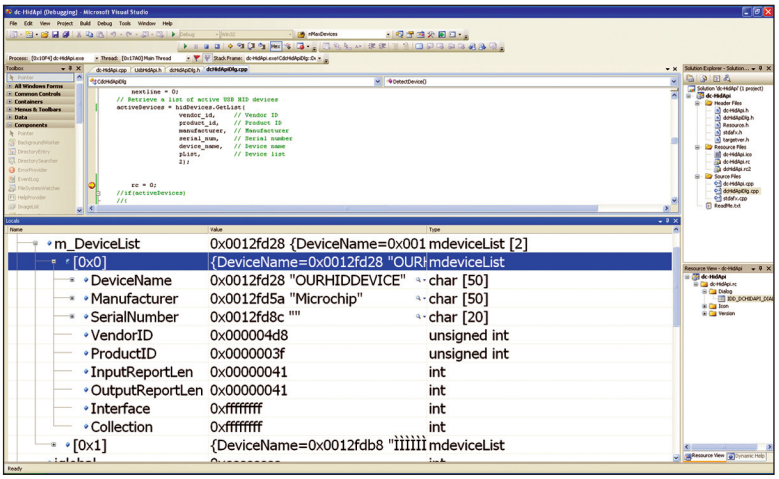
The VID entered in the descriptor can be used to trace the manufacturer of the device. However, we can also specify the manufacturer with a string in the device descriptor:

```
//Manufacturer string descriptor
ROM struct{BYTE bLength;BYTE bDscType;WORD
string[9];}sd001={
sizeof(sd001),USB_DESCRIPTOR_STRING,
{'M','i','c','r','o','c','h','i','p'
}};
```

There is no serial number entry in the descriptor. Therefore, we should expect an empty serial number structure entry to be returned from the TRAINER. If you require a serial number, the header text of the MCHPFSUSB Framework usb_descriptor.c file can be used as a guide to adding the necessary serial number descriptor code.

This whole process would be useless without a PID and VID entry. So, here are the Vendor ID (VID) and Product ID (PID) entries that I entered into the TRAINER's descriptor:

```
ROM USB_DEVICE_DESCRIPTOR device_dsc=
{
    0x12,       // Size of this descriptor
                // in bytes
    USB_DESCRIPTOR_DEVICE,
                // DEVICE descriptor type
    0x0200,     // USB Spec Release Number
                // in BCD format
    0x00,       // Class Code
    0x00,       // Subclass code
    0x00,       // Protocol code
    USB_EP0_BUFF_SIZE,
                // Max packet size for EP0,
                // see usb_config.h
    0x04D8,     // Vendor ID
    0x003F,     // Product ID
    0x0002,     // Device release number in
                // BCD format
    0x01,       // Manufacturer string index
    0x02,       // Product string index
    0x00,       // Device serial number string
                // index
    0x01        // Number of possible
                // configurations
};
```



■ SCREENSHOT 1. This is a capture of the data that flows from the PIC32MX795F512L TRAINER into the m_DeviceList structure entry. Note that the second entry is available to us but not put to use.

Since I haven't coughed up a fee for my own set of USB IDs, I've taken the liberty to use the Microchip VID coupled with a PID that is associated with the cursor-in-a-circle demo program.

If you examine the usb_descriptors.c file in the download package (available at **www.nutsvolts.com**), you'll find that only one interface and a default collection are coded. Thus, don't expect any unique collection and interface numbers to be returned. Within the collection descriptor area, you'll see that the report size is set as 64 bytes. That means 65 bytes maximum are sent along inside of a report. The 65th byte is the Report ID header byte which resides at the beginning of the report package.

Now that we know what to look for and what to expect, the results of our GetList API call can be viewed in **Screenshot 1**.

VISUAL C++ TO PIC32MX795F512L TRAINER

The TRAINER responded to the GetList API call and as a result, the variable activeDevices assumed a value of 0x0001. At this point, if multiple devices responded, we could choose the device that we wish to communicate with from our list stored in the m_DeviceList array. Since activeDevices is equal to 1, there is only one device we can talk to.

To communicate with the TRAINER, we must open the device. To do this, we use the GetList parameters returned from the TRAINER and the Kadtronix Library Open API function:

```
rc = 0;
if(activeDevices)
{
        rc = hidDevices.Open(
      pList->VendorID,
      pList->ProductID,
      pList->Manufacturer,
      pList->SerialNumber,
      pList->DeviceName,
      TRUE);    // Use non-blocking reads
}
```

We can use any of the Open API call arguments to identify the device we wish to open. Unused Open API function argument identifiers are loaded with NULL or 0xFFFF values. In our case, we're using every variable that was returned to us by the TRAINER as an identifier. The pointer *pList is currently pointing to the first entry of the m_DeviceList array. If we needed to look at the second structure in m_DeviceList, we would simply increment *pList.

I've put together a little application that will communicate its status and any data we request via a Visual C++ ListBox. Our GetList API call was successful and the variable activeDevices is used as the key to the Open API call. If the Open API call returns a TRUE (0x0001), the link to the PIC32 TRAINER was successfully opened. The code snippet that follows does all of the talking for the Open API call:

```
CString csVendorID, csProductID;
CString csInputReportLen, csOutputReportLen;
CString OPENOKtxt, OPENERRtxt;
unsigned int rc;

if (rc)
{
        OPENOKtxt.Format(_T("Device Opened"));
        m_DisplayWindow.InsertString(nextline++,
        OPENOKtxt);
        csVendorID.Format(_T("VID = 0x%0.4X"),
        pList->VendorID);
        m_DisplayWindow.InsertString(nextline++,
        csVendorID);
        csProductID.Format(_T("PID = 0x%0.4X"),
        pList->ProductID);
        m_DisplayWindow.InsertString(nextline++,
        csProductID);
        csInputReportLen.Format(_T("Input Report
        Length = %d"), pList->InputReportLen);
        m_DisplayWindow.InsertString(nextline++,
        csInputReportLen);
        csOutputReportLen.Format(_T("Output Report
        Length = %d"), pList->OutputReportLen);
        m_DisplayWindow.InsertString(nextline++,
        csOutputReportLen);
}
else
{
        OPENERRtxt.Format(_T("Device Open
        FAILED"));
        m_DisplayWindow.InsertString(nextline++,
        OPENERRtxt);
}

        return;
}
```

Screenshot 2 is a result of the positive response to the Open API function call. The Open API call did not change the contents of the m_DeviceList array. So, we can display the PIC32MX795F512L TRAINER GetList values with our positive link message.

I failed to mention that I had a little HID analog-to-digital (A-to-D) conversion routine in my pocket. So, I'm

sure you're wondering what clicking on the Read Voltages button does. Once again, I'll shut up and just show you the code:

```
void CdcHidApiDlg::OnGetVolts()
{
        char xmit_buf[100];
        char recv_buf[100];
        unsigned int wrc,rrc;
        float raw_volts;
        CString vReading;

        memset(xmit_buf, 0,
        sizeof(xmit_buf));
        xmit_buf[1] = 0x37;
        wrc = hidDevices.Write(xmit_buf);
```

The partial code snippet of the OnGetVolts function creates a pair of 100 byte buffers. The xmit_buf array is preloaded with zeros which automatically puts our desired Report ID (0x00) in the first report buffer slot. The Report ID is immediately followed by a command byte of 0x37 which corresponds to a C case statement in the download package file nv-pic32mx-HID.c:

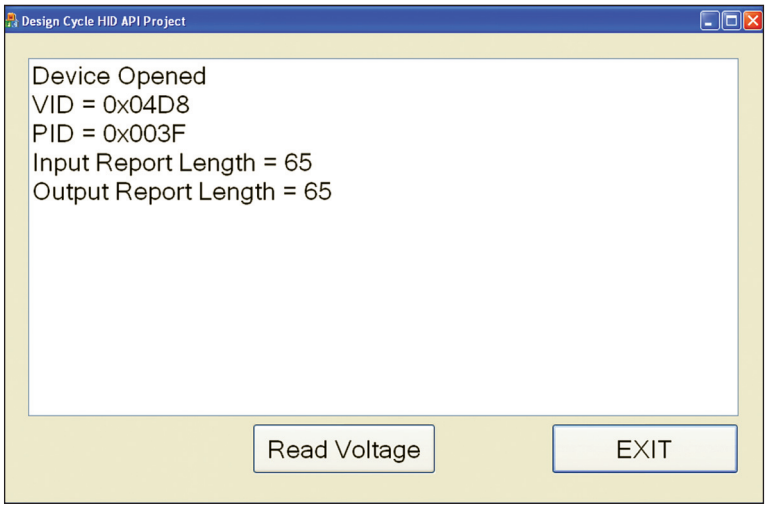
```
case 0x37:     //Read POT command
  {
    WORD_VAL w;

    if(!HIDTxHandleBusy(USBInHandle))
        {
           mInitPOT();
           w = ReadPOT();
             // Use ADC to read the
             // I/O pin voltage.
           ToSendDataBuffer[0] = 0x37;
             // Echo back to the host
           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;
```

The ReadPOT function is configured to use RB2 as the analog input for the A-to-D converter. As you can see in **Schematic 1**, I've tied the wiper of a 10K potentiometer to the PIC32MX795F512L's RB2 pin. With this potentiometer configuration, the PIC32's A-to-D converter input will be presented with a minimum of zero volts and a maximum of 3.3 volts. Following the A-to-D conversion, the command byte and the raw voltage word are returned to the host's Visual C++ application. The modified hardware to support **Schematic 1** is lying under the lens in **Photo 1**.

PIC32MX795F512L TRAINER TO VISUAL C++

If all has gone as planned, the TRAINER received the 0x37 command and executed the A-to-D converter

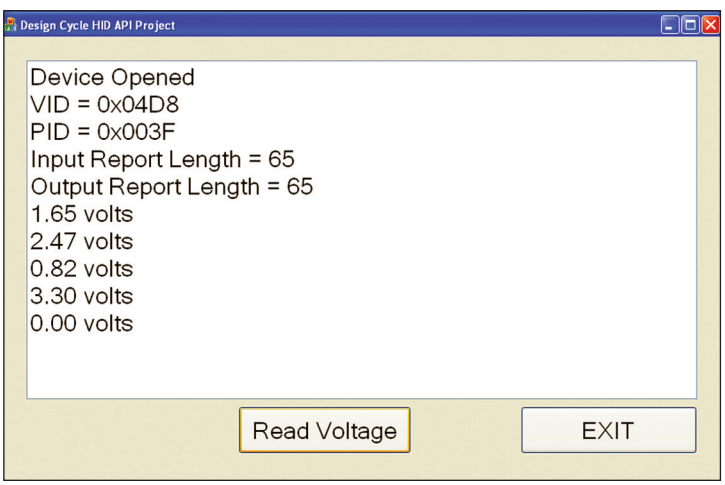


■ SCREENSHOT 2. Nothing fancy here. This is a simple Visual C++ ListBox control and a couple of buttons. The heavy lifting is being done by the Kadtronix USB HID API Library and some user-contributed Visual C++ event handlers.

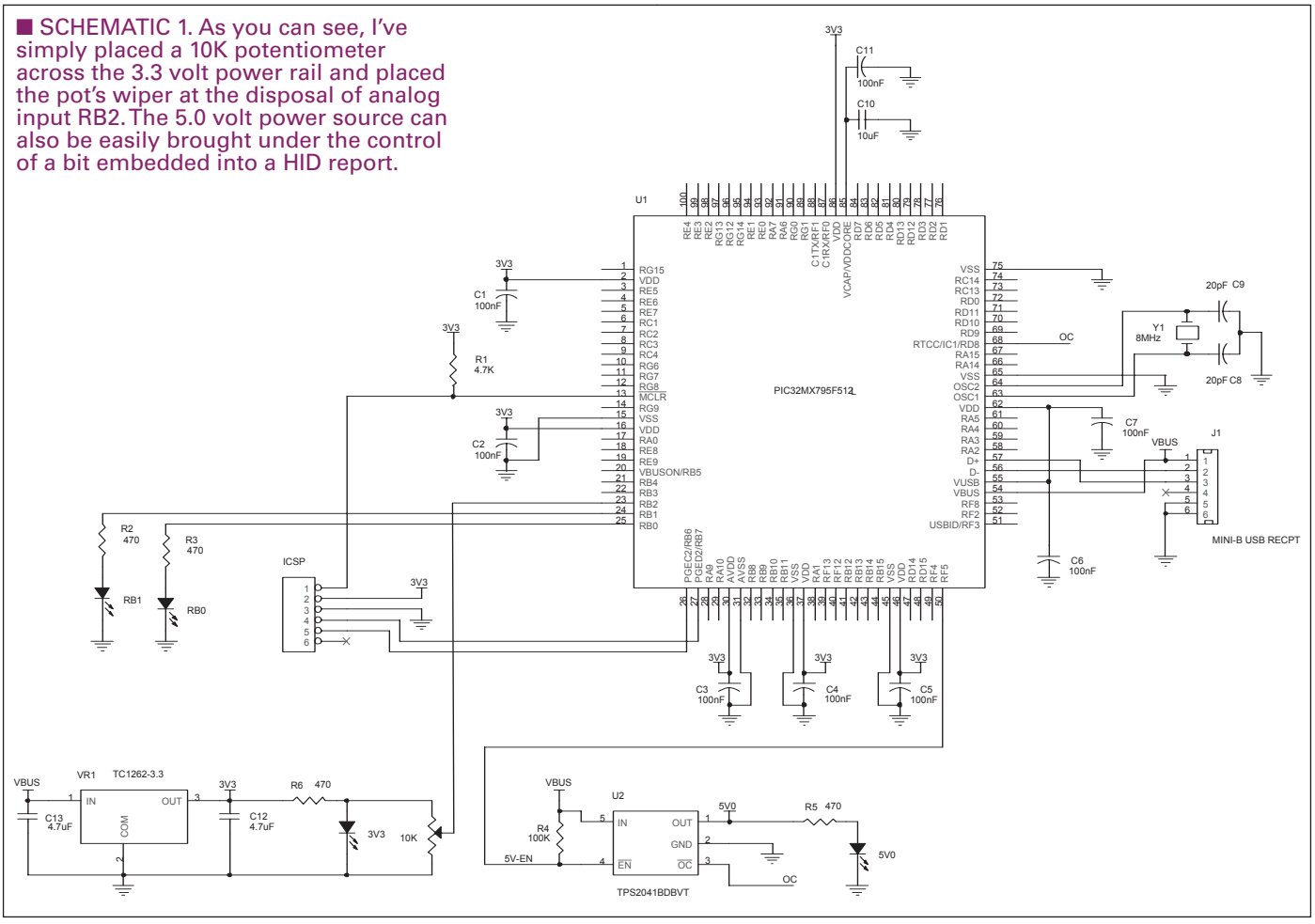
read operation against the potentiometer which is configured as a simple voltage divider. Approximately 100 mS later, the host issues a Read API function call to the open device which just happens to be our TRAINER. Here's the receive portion of the OnGetVolts function:

```
if (wrc == hidDevices.m_WriteSize)
{
  Sleep(100);
  rrc = hidDevices.Read(recv_buf);
  if(rrc == hidDevices.m_ReadSize)
       {
       raw_volts = (float)((recv_buf[2] >> 8)&
       0x00FF) + ((recv_buf[3] << 8) & 0xFF00);
       vReading.Format(_T("%2.2f
       volts"),raw_volts * .003222);
       m_DisplayWindow.InsertString
       (nextline++, vReading);
       }
}
}
```

■ SCREENSHOT 3. Each click returns a voltage reading. Everything in the ListBox with the exception of the Device Opened message is the direct result of HID report data flowing between the host Visual C++ application and the PIC32MX795F512L TRAINER.

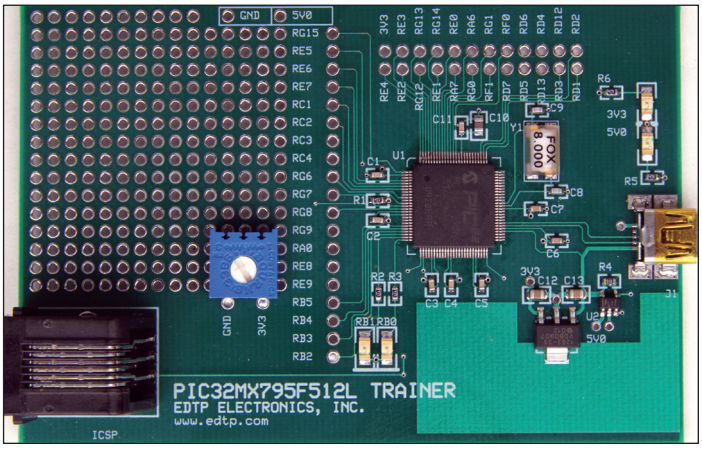


■ SCHEMATIC 1. As you can see, I've simply placed a 10K potentiometer across the 3.3 volt power rail and placed the pot's wiper at the disposal of analog input RB2. The 5.0 volt power source can also be easily brought under the control of a bit embedded into a HID report.



The Report ID byte and the echoed command byte precede the actual raw voltage data in the receive buffer. Since the 10 bits of incoming raw voltage data is in byte format, we need to convert the pair of voltage bytes to an integer. In that we must do some scaling of the integer voltage value for display in the ListBox, the integer voltage value must be converted to a floating point value. Twisting the potentiometer's wiper and clicking on the Read Voltage button produced the content contained within **Screenshot 3**.

■ PHOTO 1. This is a shot of my slightly modified PIC32MX795F512L TRAINER. The voltage from the potentiometer's wiper is one of many data types that can be transferred in a HID report. If you send a 0x80 instead of the 0x37, you'll gain control of the LEDs attached to RB0 and RB1.



AS EASY AS RS-232

I never thought I would admit it. With the MCHPFSUSB Framework working on the TRAINER side and the Kadtronix Library operating within Microsoft's Visual C++ component of VS 2008, coding a useful HID

SOURCES

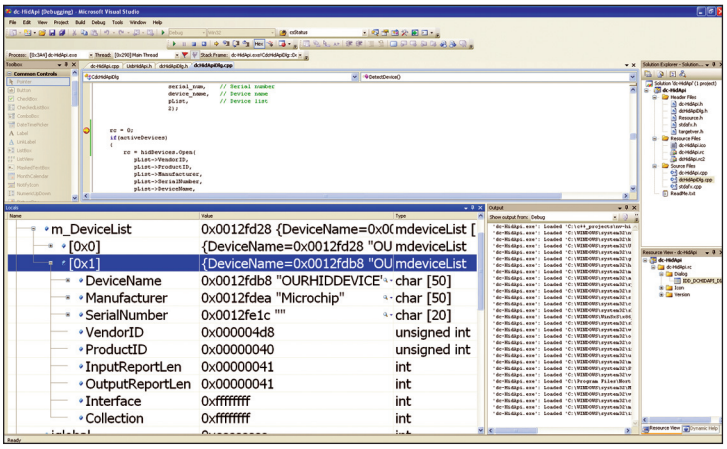
Microchip
MCHPFSUSB Framework ; PIC32MX795F512L
www.microchip.com

Microsoft
Visual Studio 2008
www.microsoft.com

Kadtronix
Kadtronix USB HID API Library
www.kadtronix.com

EDTP Electronics, Inc.
PIC32MX795F512L TRAINER
www.edtp.com

■ SCREENSHOT 4. We can easily add and communicate with more than one PIC32MX795F512L TRAINER by simply increasing the number of m_DeviceList entries to accommodate the number of uniquely identified TRAINERs.



data transfer application is just as easy as coding a similar RS-232 based program.

I couldn't leave you without putting a second PIC32MX795F512L TRAINER out there and trying to contact it. So, I changed the PID to 0x0040 and programmed a second TRAINER. You can see its second m_DeviceList entry in **Screenshot 4**. Boy, does that conjure up possibilities. I can see all of those light bulbs illuminating over your heads.

I'll put all of the good hicker nuts I found into the download package. Once you ramble through the basket of nuts, you can add host and HID-class device programming to your Design Cycle. **NV**

■ *Fred Eady can be contacted via email at fred@edtp.com and the EDTP Electronics website at* ***www.edtp.com****.*