# Implementación de Comunicación USB con Microcontrolador PIC18F4550 y LabVIEW

Lic. Física Yohan Pérez-Moret
PCB por La Derecha
contacto@pcbporladerecha.com

## 1. Resumen

Se presenta el desarrollo, simulación e implementación de la comunicación USB con un microcontrolador PIC18F4550 y LabVIEW. El código programado al microcontrolador PIC envía cíclicamente la lectura de su conversor análogo-digital (AD) y el estado del pin RA4 a una computadora personal a través del bus USB. El código del microcontrolador fue escrito en lenguaje C, con el compilador CCS 4.018. En LabVIEW 8.0 se generó el driver para el dispositivo USB y se diseñó un instrumento virtual (VI) para atender la comunicación. El VI se encarga de graficar la lectura AD del PIC y a petición del usuario que lo opera de enviarle un valor de un byte, el cual es cargado en el puerto B del PIC. Se realiza la simulación utilizando Proteus 7.2 y el VI diseñado. Por último se muestra la implementación práctica en la placa de demostración PICDEM 2 plus, la cual fue modificada para la funcionalidad USB.

**Palabras claves:** USB, microcontrolador PIC, LabVIEW, PICC, MPLAB, Instrumentación, PICDEM.

## 2. Introducción

El bus serie universal o en sus siglas en inglés: USB, posee algunas características como son:

- Integridad de la señal por el uso de apantallamientos, drivers y receptores diferenciales.
- Los dispositivos USB de clase HID (Human Interface Device) son soportados por los sistemas operativos desde Windows Millennium en adelante.
- En dispositivos USB 2.0 se pueden transmitir datos a razón de 480 megabits por segundo.
- Los cables USB individuales pueden extenderse hasta 5 m de distancia y más con el uso de multiplexores de puertos o HUB.
- El bus USB puede suplir hasta 500 mA @ 5 V a cada uno de los dispositivos conectados, eliminando el uso de fuentes y cables externos en aplicaciones de baja potencia.

Esas características, por citar solo algunas, hacen del bus USB una opción útil en aplicaciones de instrumentación electrónica. Muchos fabricantes de microcontroladores programables como Microchip, están incluyendo un modulo USB en sus dispositivos. Facilitando el desarrollo de aplicaciones de instrumentación que aprovechen las capacidades USB.

Por otra parte, el soporte brindado por los fabricantes de computadoras personales (PC) a los puertos seriales RS-232 y paralelos Centronics, de amplio uso entre los instrumentistas, cada vez es menor. En la actualidad es común que una PC incluya solo un puerto serial RS-232, o ninguno en el caso de algunas portátiles, y sí varios puertos USB.

En el presente trabajo se presenta el uso del módulo USB que posee el mi-

crocontrolador PIC18F4550 (1) del fabricante Microchip y su atención utilizando LabVIEW 8.0.

**2.1. Estructura USB**

La funcionalidad de los dispositivos USB está estructurada en capas, véase la Fig. 1. La capa de mayor jerarquía, después del dispositivo en sí, es la de configuración. Un dispositivo puede tener múltiples configuraciones. Por ejemplo, un dispositivo puede tener varias exigencias de energía según el modo en que esté: auto-energizado o bus-energizado.

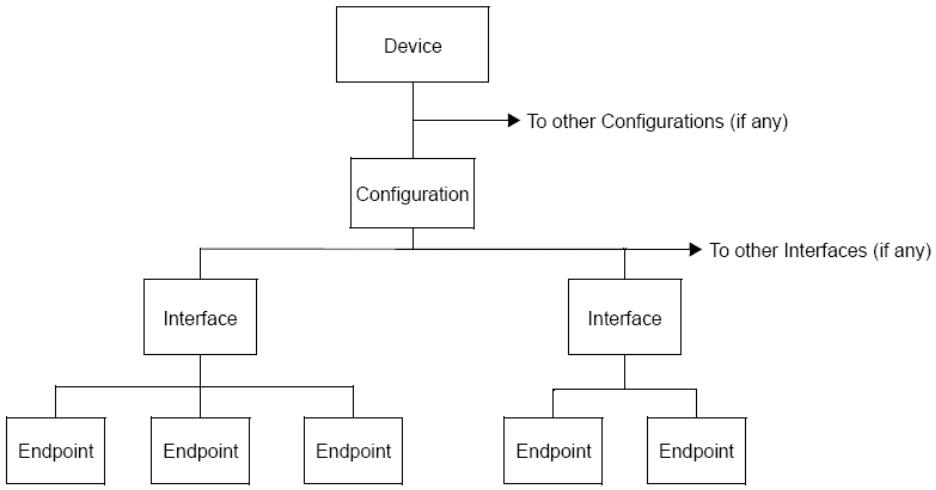


Fig. 1. Estructura de capas del USB

Por cada capa de configuración pueden existir múltiples capas de interfaces. Por debajo de la interfaz están los endpoints. Los datos son transferidos directamente a ese nivel. El endpoint-0 es siempre de control y cuando un dispositivo se conecta al bus debe estar disponible.

La información comunicada al bus está agrupada en paquetes temporales de 1 ms conocidos como frames. Cada frame puede contener tantas transacciones como dispositivos y enpoints estén conectados.

**Transferencias**

Hay cuatro tipos de transferencias especificadas por la norma USB (2).

- **Isocrónica:** Provee un método para transferir grandes cantidades de datos (hasta 1023 bytes) con una temporización de envío asegurada (isocrónica: de igual tiempo); aunque la integridad de los datos no se asegura. Utilizado en aplicaciones de transmisión continua (streaming) y donde pequeñas pérdidas de datos no son críticas, como en el caso de transmisión de audio.
- **Bulk:** Este método de transferencia permite el envío de grandes cantidades de datos, asegurando su integridad. Pero no se garantiza la temporización entre envíos.
- **Interrupción:** Este método asegura la temporización y la integridad de los datos para pequeños bloques de datos.
- **Control:** Este tipo de transferencia es para configuración y control del dispositivo conectado al bus USB.

Los dispositivos de alta velocidad soportan todos los tipos de transferencia; el resto está limitado a transferencias por interrupción y control.

**3. Materiales y Métodos**

Para la implementación y montaje de la comunicación USB se utilizaron las siguientes herramientas:

- compilador de C PIC-C versión 4.018 (3): para la escritura del código fuente y la generación del archivo de programación HEX del PIC.
- Proteus 7.2 (4): entorno de captura y simulación de circuitos electrónicos. La versión 7.2 incluye librerías para el microcontrolador PIC18F4550 y la simulación USB.
- LabVIEW 8.0: Entorno de programación gráfica.
- PICDEM 2 Plus de Microchip: Placa de demostración para el trabajo con los microcontroladores de la gama media y alta. Posee módulos de hardware prediseñados de alimentación, visualización con LCD y LED, sockets de 18, 28 y 40 pines así como terminales para fácil acceso a los puertos.
- Cable de conexión USB tipo "A plug" para conexión del PIC a la PC.
- Microcontrolador PIC18F4550

Si no se dispone de recursos materiales como el PIC18F4550 y el PICDEM 2 Plus, el trabajo puede realizarse hasta la etapa de simulación con el uso de PICC y Proteus.

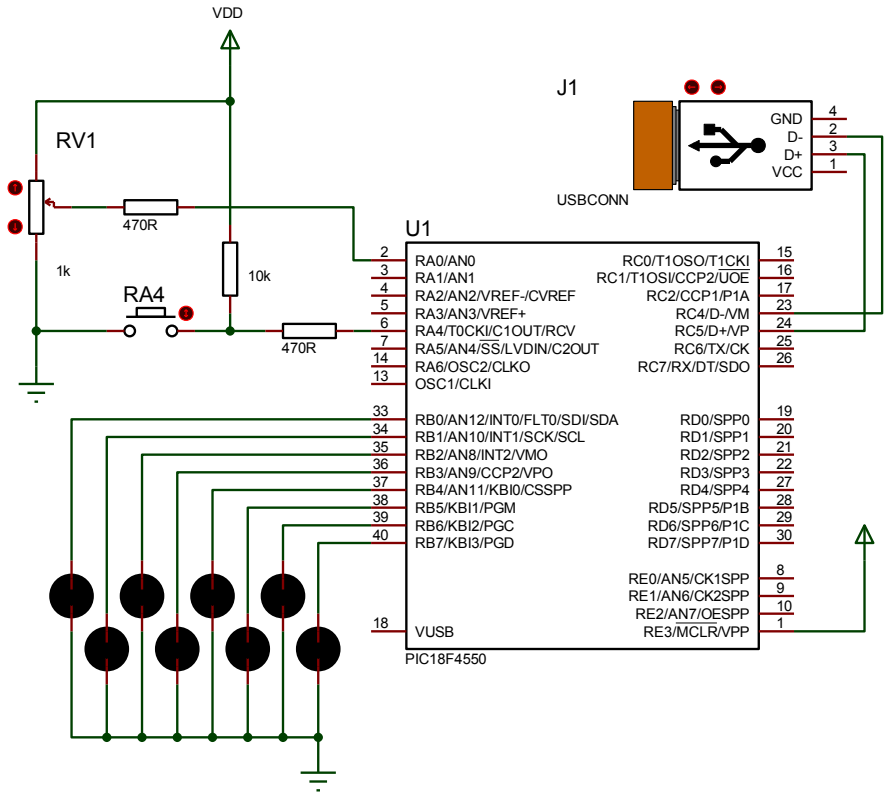


Fig. 2. Circuito diseñado en Proteus solo para fines de simulación USB con el PIC18F4550

En Proteus se montó el circuito de la Fig. 2 con el fin de simular el comportamiento del microcontrolador PIC18F4550 y el bus USB. Al puerto B del PIC se conectaron diodos LED para visualizar el dato de tipo byte recibido vía USB.

El PIC fue programado en Proteus con el archivo tipo COF generado durante la compilación del código fuente. Este último se muestra en el anexo A y al cual volveremos más adelante.

### 3.1. Configuración del Reloj del PIC para el uso del módulo USB

El PIC18F4550 incluye un sistema de generación de reloj distinto a los microcontroladores de su misma gama. Ello para poder cumplir los requerimientos del USB en alta y baja velocidad. El PIC18F4550 incluye un sistema de prescalers y postscalers que garantizan la frecuencia del módulo USB en alta y baja velocidad a partir de diferentes valores de un oscilador primario. Sucede que el módulo USB solo acepta 6 MHz para el modo de baja velocidad o 48 MHz para alta velocidad, respectivamente. Por ello, el microcontrolador ofrece un juego de fusibles, que bien configurados, permiten lograr dichas velocidades a partir de otras frecuencias del reloj externo utilizado. En la sección 2.0 de la referencia (1) se dan los detalles sobre la configuración del reloj del módulo USB.

Cuando se diseña el código fuente para usar el módulo USB del PIC18F4550, en la línea que define la frecuencia del oscilador se debe incluir la frecuencia a usar en el núcleo del microcontrolador, que no necesariamente tiene que coincidir con la del oscilador utilizado. Por ejemplo, en nuestro caso se empleó un oscilador activo de 4 MHz, no obstante, en la línea 39 del código fuente, ver anexo A, se definió otro valor de frecuencia: 24 MHz, el cual corresponde a la frecuencia a la que trabajará el núcleo del microcontrolador al ser activado el PLL multiplicador de frecuencia, necesario en este caso para generar la frecuencia correcta al módulo USB.

Luego, en la línea 40, se programaron los fusibles USBDIV, PLL y CPUDIV con valores tales que garanticen que a partir de los 4 MHz del oscilador externo, el módulo USB trabaje a la frecuencia de 48 MHz definida en la línea 39.

Es importante señalar también que para utilizar el módulo USB no cualquier valor de reloj es válido. En la sección 2.3 de la referencia (1) se muestra una tabla con las frecuencias y tipos de osciladores compatibles con el USB, relacionados con los valores de los fusibles de configuración USBDIV, PLL y CPUDIV. Una sección de esa tabla se reproduce en la Fig. 3.

Por ejemplo, si se dispone del oscilador HS de 24 MHz de la columna "Input Oscillator frecuency" de la Fig. 3, entonces las frecuencias posibles para el núcleo del microcontrolador serían 24 MHz, 12 MHz, 8 MHz y 6 MHz, según la columna "Microcontroller Clock Frecuency". Suponiendo que escogemos la frecuencia de 8 MHz para el núcleo tendríamos que escribir las siguientes líneas de código en PICC:

```
#fuse delay(clock=8000000)
#fuses HS, NOWDT, NOPROTECT, NOLVP,
NODEBUG, USBDIV, PLL6, CPUDIV3,
VREGEN
```

Nótese que en la primera línea de código anterior, no se definió la frecuencia del oscilador externo HS de 4 MHz, sino la frecuencia de trabajo resultante del PLL y CPUDIV escogidos.

| Input Oscillator Frequency | PLL Division (PLLDIV2:PLLDIV0) | Clock Mode (FOSC3:FOSC0) | MCU Clock Division (CPUDIV1:CPUDIV0) | Microcontroller Clock Frequency |
|---|---|---|---|---|
| 48 MHz | N/A(1) | EC, ECIO | None (00) | 48 MHz |
| | | | +2 (01) | **24 MHz** |
| | | | +3 (10) | 16 MHz |
| | | | +4 (11) | 12 MHz |
| 48 MHz | +12 (111) | EC, ECIO | None (00) | 48 MHz |
| | | | +2 (01) | **24 MHz** |
| | | | +3 (10) | 16 MHz |
| | | | +4 (11) | 12 MHz |
| | | ECPLL, ECPIO | +2 (00) | 48 MHz |
| | | | +3 (01) | 32 MHz |
| | | | +4 (10) | **24 MHz** |
| | | | +6 (11) | 16 MHz |
| 40 MHz | +10 (110) | EC, ECIO | None (00) | 40 MHz |
| | | | +2 (01) | 20 MHz |
| | | | +3 (10) | 13.33 MHz |
| | | | +4 (11) | 10 MHz |
| | | ECPLL, ECPIO | +2 (00) | 48 MHz |
| | | | +3 (01) | 32 MHz |
| | | | +4 (10) | **24 MHz** |
| | | | +6 (11) | 16 MHz |
| 24 MHz | +6 (101) | HS, EC, ECIO | None (00) | **24 MHz** |
| | | | +2 (01) | 12 MHz |
| | | | +3 (10) | 8 MHz |
| | | | +4 (11) | 6 MHz |
| | | HSPLL, ECPLL, ECPIO | +2 (00) | 48 MHz |
| | | | +3 (01) | 32 MHz |
| | | | +4 (10) | **24 MHz** |
| | | | +6 (11) | 16 MHz |

Fig. 3. Selección de la frecuencia USB a partir de la frecuencia del oscilador de entrada

### 3.2. Descriptores USB

Cuando un dispositivo USB es conectado a la PC el sistema operativo es capaz de reconocer su fabricante, identificador de producto e instalarle un driver estándar HID o abrir un asistente para encontrar el adecuado. También muestra mensajes sobre el estado del dispositivo, marca, modelo, etc. Toda esa información que requiere el sistema operativo para identificar al dispositivo USB está almacenada en el microcontrolador PIC, en una zona de memoria RAM USB. En particular en el banco 4 de la RAM USB, destinado al buffer de descriptores.

Los descriptores son archivos que se incluyen junto al código fuente para ser grabados a esa zona de memoria RAM USB. El diseño de un descriptor es el tema de mayor complejidad en el uso de la comunicación USB. Las referencias (2; 5) brindan información sobre el diseño de descriptores. En nuestro caso tomamos y adaptamos el descriptor brindado en uno de los ejemplos USB del compilador CCS los cuales también son una referencia de consulta válida.

El anexo B muestra el descriptor agregado al código fuente del microcontrolador. La inclusión del descriptor en el archivo fuente se realizó a través de la directiva “include” de la línea 64 de este último.

En el descriptor se incluyen el identificador del fabricante del dispositivo (VID) y el identificador del producto (PID). Esos identificadores son utilizados por el sistema operativo de la PC para encontrar el driver apropiado al dispositivo USB. Las líneas 184 y 185 del descrip-

tor, véase el anexo B, contienen el VID y el PID, respectivamente. La línea 131 declara el tipo de transferencia que soportará el dispositivo, en este caso el código significa transferencia por interrupción.

### 3.3. Driver e instrumento Virtual en LabVIEW para el dispositivo USB

Para que nuestro dispositivo USB pueda ser controlado desde LabVIEW es necesario que este lo reconozca como propio, con un driver USB desarrollado en LabVIEW. Para ello LabVIEW brinda una utilidad llamada "VISA Driver Development Wizard – Hardware Bus", véase la Fig. 4. Esta da la posibilidad de desarrollar un driver para dispositivos PCI, USB o FireWire.

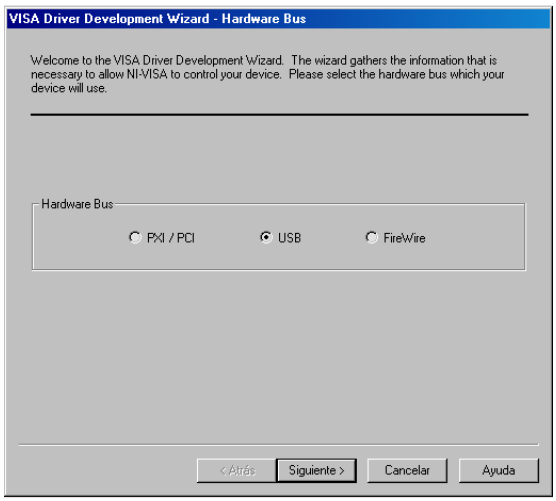


Fig. 4. Utilidad para desarrollar Driver USB de LabVIEW

En el siguiente paso del mismo asistente se encuentran los campos del VID y del PID. Estos campos se llenan en correspondencia con las líneas 184 y 185 del descriptor del anexo B, tal y como se muestra en la Fig. 5.

Después de especificar el PID y el VID el asistente creará un archivo de tipo INF en el directorio y nombre especificados en el paso mostrado en la Fig. 6.

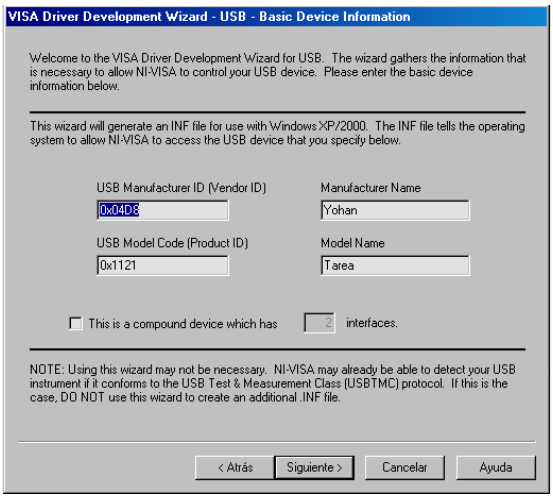


Fig. 5. Especificación del VID y del PID del driver

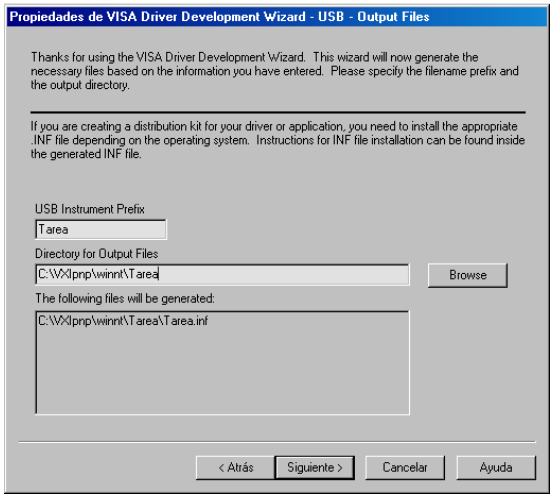


Fig. 6. Finalización del asistente, nombre y lugar del driver .inf a crear.

Con el código del microcontrolador listo y el driver en LabVIEW disponible, podemos iniciar la simulación del microcontrolador USB y observar cómo este es detectado por el sistema operativo.

#### 3.3.1. Simulación del dispositivo con Proteus

Al iniciar la simulación del circuito de la Fig. 2 el sistema operativo detecta la presencia de un dispositivo USB, iniciando el proceso de "enumeración" que básicamente consiste en asignarle una dirección en el bus al dispositivo. Después de esto se abre el asistente para Hardware nuevo encontrado, tal y como se muestra en la Fig. 7. Con este asistente

se localiza el driver INF creado en LabVIEW, instalándolo desde una lista o ubicación específica, Fig. 8.

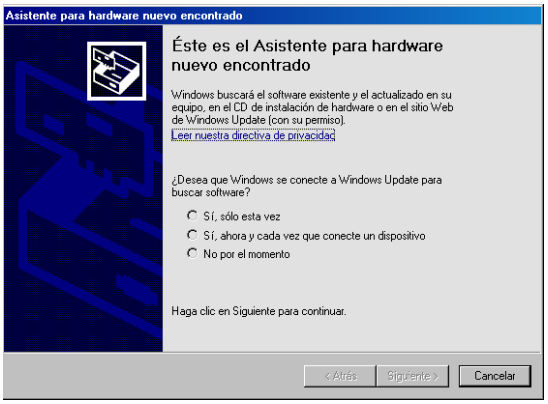


Fig. 7. Asistente para localizar el driver del dispositivo

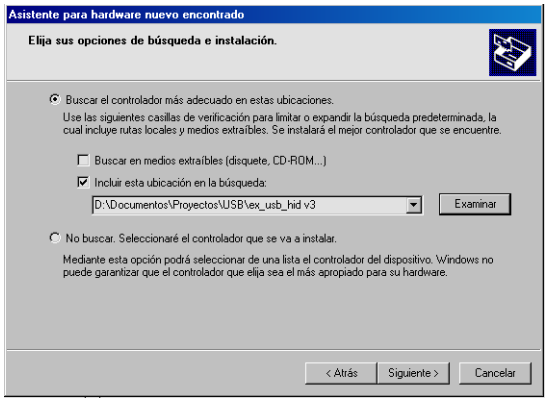


Fig. 8. Localización del driver INF

Después del paso anterior el asistente concluye de instalar el driver para el dispositivo, Fig. 9 y Fig. 10.

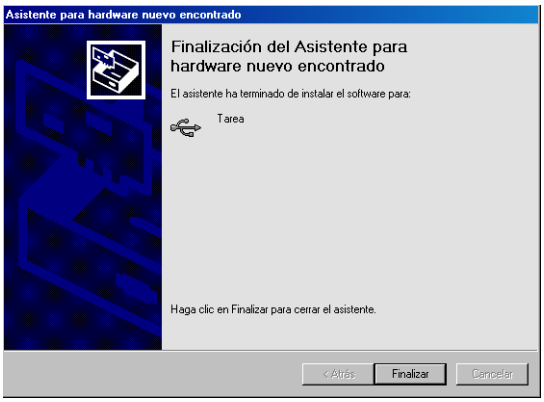


Fig. 9. Finalización del asistente

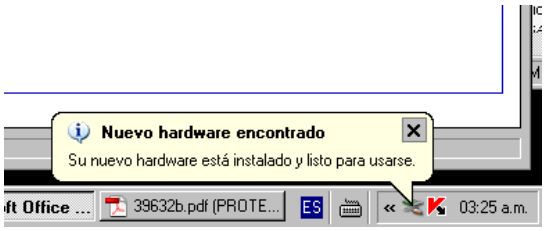


Fig. 10. Confirmación de Windows

Si ahora observamos el administrador de Hardware de Windows XP podremos percatarnos de que se ha agregado una nueva categoría, la categoría de dispositivos USB de National Instruments (NI-VISA USB). Y bajo esta última aparecerá nuestro dispositivo USB. Ello significa que a partir de ahora LabVIEW reconoce nuestro dispositivo como propio y estaremos en condiciones de controlarlo con un VI, empleando las VISAS de LabVIEW. La Fig. 11 muestra la categoría de NI-VISA USB Devices en el administrador de dispositivos de Windows XP.

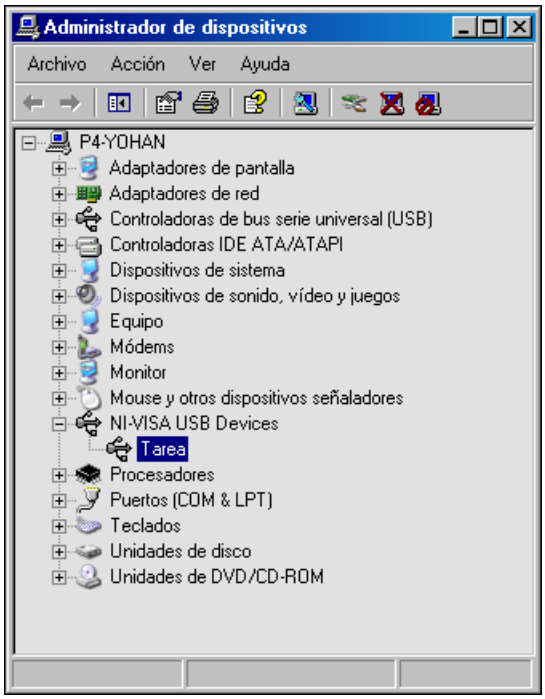


Fig. 11. Presencia del dispositivo en el Administrador de Dispositivos de Windows

### 3.3.2. Instrumento Virtual

Antes de pasar al desarrollo de un instrumento virtual o VI para la comunicación con nuestro dispositivo USB es conveniente obtener su "resource name". Ello se puede hacer ejecutando la utilidad de LabVIEW llamada "Visa Interactive Control", mostrada en la Fig. 12. De ella podremos determinar que el "VISA resource name" de nuestro dispositivo es la cadena: "USB0::0x04D8::0x1121::NI-VISA-0::RAW", con ella podremos referenciar a nuestro dispositivo desde un VI.

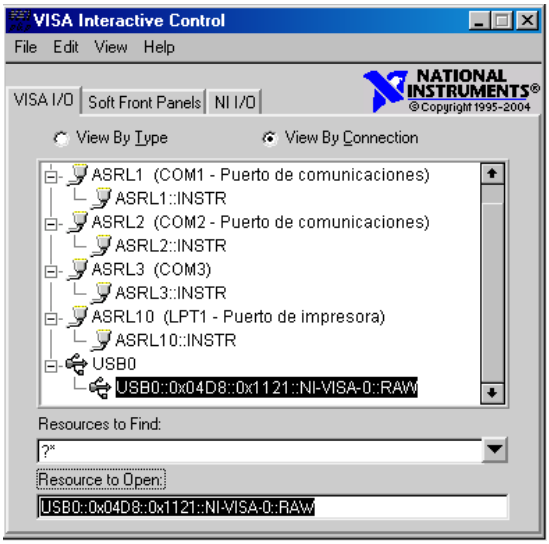


Fig. 12. Visa Interactive Control para determinar el VISA Resource Name

Nótese como en el "resource name" del dispositivo aparecen los códigos PID y VID que habíamos programado al microcontrolador en su descriptor. En la Fig. 13 se observa el panel frontal del VI desarrollado. Al fondo de este el circuito desarrollado en Proteus interactuando con el VI. Los cambios realizados en el potenciómetro RV1 eran registrados por el instrumento virtual cíclicamente y al activar el botón LEDs del Puerto B del VI el circuito respondía poniendo en alto todos los pines del puerto B, se enviaba el código hexadecimal 0xFF hacia el PIC o el código 0x00 en caso de desactivar el botón de Activar LEDs del VI.

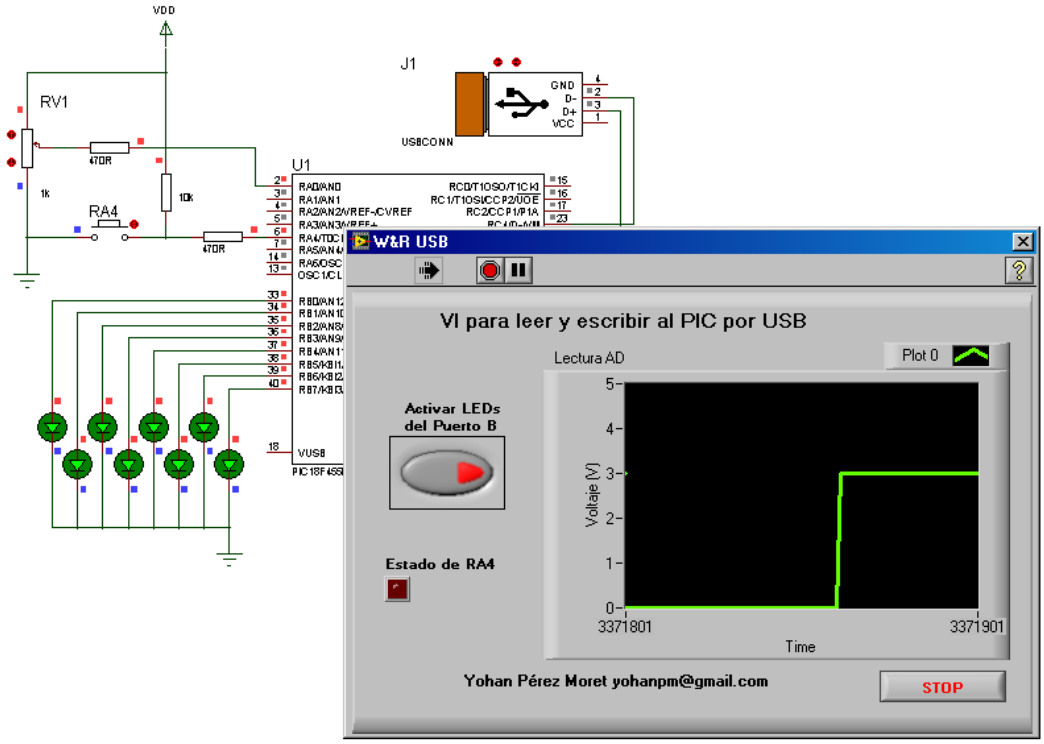


Fig. 13. Panel Frontal del VI desarrollado y simulación en Proteus

La ventana de código del VI desarrollado se muestra en la Fig. 14. Se programaron dos ciclos independientes, uno para el envío y otro para la recepción de las lecturas AD del PIC. La llegada de un dato al puerto USB de la PC es detectada por el VI a través de una interrupción USB.

El envío y la recepción se realizan independientemente la una de la otra, full dúplex.

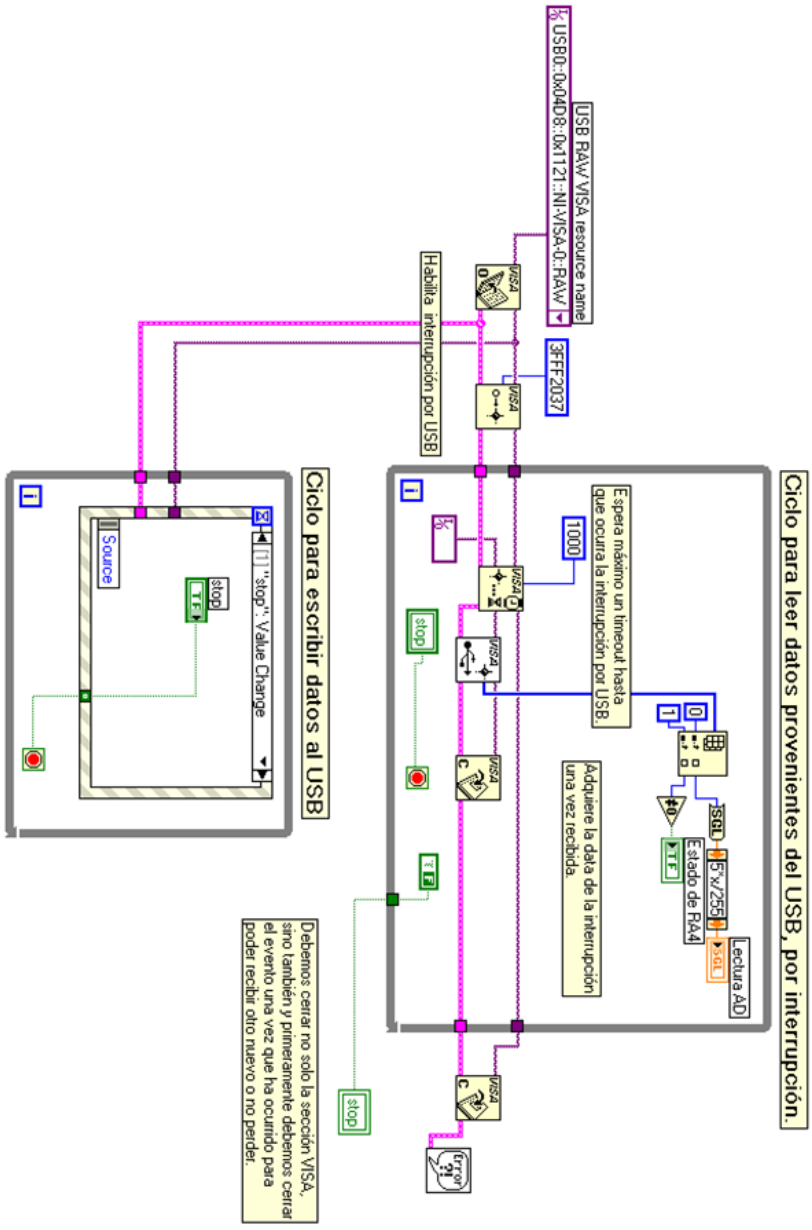


Fig. 14. Diagrama de código del VI desarrollado

## 4. Implementación Práctica

Para el montaje práctico del PIC18F4550 se adaptó el PICDEM 2 Plus, incluyéndole un conector Header de 4 pines en su área de prototipo. Los pines 2 y 3 del conector Header se unieron a través de cables con los pines 23 y 24 del socket de 40 pines del PICDEM, respectivamente. Los pines 23 y 24 se corresponden con las señales diferenciales D- y D+ respectivamente. El pin 1 del conector Header se conectó al común del PICDEM 2 plus y el pin 5 al terminal de 5 V de la misma placa de demostración, con el objetivo de alimentarla directamente del bus USB.

Los resultados obtenidos con el montaje práctico fueron los mismos que con la simulación en Proteus 7.2. El VI diseñado se comportó de igual manera y sí se notó la diferencia de velocidad entre el montaje práctico y la simulación, esta última más lenta que la primera. Al conectar el dispositivo USB por primera vez a la PC Windows ya lo reconocía y pudo cargar el driver correspondiente que previamente había sido instalado en la fase de simulación con Proteus.

## 5. Conclusiones

Se logró realizar la simulación y la implementación práctica de la comunicación USB utilizando un microcontrolador PIC18F4550 y un VI diseñado en LabVIEW.

El desarrollo de los descriptores USB aún debe profundizarse ya que estos constituyen una gran parte de la configuración del dispositivo USB.

## 7. Referencias


1. **Inc., Microchip Technology.** *PIC18F2455/2550/4455/4550 Data Sheet.* s.l. : Microchip, 2004. DS39632B.

2. USB Specification, version 2.0: Chapter 9. [Online] http://www.usb.org.

3. **CCS.** [Online] http://www.ccsinfo.com/.

4. [Online] http://www.labcenter.co.uk/.

5. **Condit, Reston.** *TB054: An Introduction to USB Descriptors with a Game Port to USB Game Pad Translator Example.* s.l. : Microchip Technology, 2004. DS91054C.

6. **Dearborn, Scott.** *AN971: USB Port-Powered Li-Ion/Li-Polymer Battery Charging.* s.l. : Microchip Technology Inc, 2005.

7. **Condit, Reston.** *AN258: Low Cost USB Microcontroller Programmer.* s.l. : Microchip Technology Inc., 2003. DS00258A.

8. **Rojvanit, Rawin.** *AN956: Migrating Applications to USB from RS-232 UART with Minimal Impact on PC software.* s.l. : Microchip Technology, 2004. DS00956B.

9. **Microchip.** [Online] http://www.microchip.com.

## Anexos

## Anexo A. Código Fuente en lenguaje C del PIC18F4550 con USB

```
1  ////////////////////////////////////////////////////////////////////////////
2  ////                                                                    ////
3  ////                            usb_hid.c                               ////
4  ////                                                                    ////
5  //// A simple demo that shows how to use the HID class to send        ////
6  //// and receive custom data from the PC.  Normally HID is used       ////
7  //// with pre-defined classes, such as mice and keyboards, but        ////
8  //// can be used for vendor specific classes as shown in this         ////
9  //// example.                                                         ////
10 ////                                                                    ////
11 ////////////////////////////////////////////////////////////////////////////
12
13
14 ////////////////////////////////////////////////////////////////////////////
15 ///                                                                      ///
16 /// En 1 para activar modulo USB, en 0 para usar un periférico           ///
17 /// USBN960X. El PIC18F4550 tiene módulo USB por lo que no hay           ///
18 /// que ponerlo en 0.                                                    ///
19 ///                                                                      ///
20 ////////////////////////////////////////////////////////////////////////////
21 #define __USB_PIC_PERIF__ 1      //Activar módulo USB                    ///
22 ////////////////////////////////////////////////////////////////////////////
23
24
25 #define USB_USE_FULL_SPEED  FALSE    //Trabajaremos USB 1.0.
26 #include <18F4550.h>                 //El código será para un PIC18F4550
27
28 ////////////////////////////////////////////////////////////////////////////
29 ///                                                                      ///
30 /// La frecuencia de reloj que se define no es la del                    ///
31 /// cristal externo, como siempre se hace, sino que se pone el que       ///
32 /// queremos para el USB del PIC. Con la PLL del PIC se logrará llevar   ///
33 /// el reloj externo al valor que fijemos para USB. Por eso hay que      ///
34 /// configurar la PLL y el divisor de CPU con valores tales que:         ///
35 /// dado un reloj externo se obtenga el reloj de USB que se fija a       ///
36 /// continuación. En la datasheet del PIC hay una tabla con las          ///
37 /// combinaciones posibles de PLL, CPUDIV y CLOCKs de USB y externo.     ///
38 ///                                                                      ///
39 ////////////////////////////////////////////////////////////////////////////
40 #use delay(clock=24000000)  //Reloj para USB (¡No reloj del PIC!)       ///
41 #fuses HSPLL,NOWDT,NOPROTECT,NOLVP,NODEBUG,USBDIV,PLL1,CPUDIV3,VREGEN
//Lo anterior incluye la Configuración del módulo USB, pll, cpudiv y usbdiv.
```

```
42
43    //Tells the CCS PIC USB firmware to include HID handling code.
44    #define USB_HID_DEVICE  TRUE
45
46    //the following defines needed for the CCS USB PIC driver to enable the TX endpoint 1
47    // and allocate buffer space on the peripheral
48    #define USB_EP1_TX_ENABLE  USB_ENABLE_INTERRUPT   //turn on EP1 for IN bulk/interrupt transfers
49    #define USB_EP1_TX_SIZE    8  //allocate 8 bytes in the hardware for transmission
50
51    //the following defines needed for the CCS USB PIC driver to enable the RX endpoint 1
52    // and allocate buffer space on the peripheral
53    #define USB_EP1_RX_ENABLE  USB_ENABLE_INTERRUPT   //turn on EP1 for OUT bulk/interrupt transfers
54    #define USB_EP1_RX_SIZE    8  //allocate 8 bytes in the hardware for reception
55
56    ////////////////////////////////////////////////////////////////////////////
57    //
58    // Include the CCS USB Libraries.  See the comments at the top of these
59    // files for more information
60    //
61    ////////////////////////////////////////////////////////////////////////////
62
63    #include <my_pic18_usb.h>       //Microchip 18Fxx5x hardware layer for usb.c
64    #include <NI_usb_desc_hid.h>    //Descriptor HID que usa el driver de LabVIEW .inf
65    #include <my_usb.c>             //handles usb setup tokens and get descriptor reports
66
67    ////////////////////////////////////////////////////////////////////////////
68    //
69    // Configure the demonstration I/O
70    //
71    ////////////////////////////////////////////////////////////////////////////
72
73    #define BUTTON PIN_A4
74
75    void main() {
76       int8 out_data[20];   //Buffer para datos que serán enviados.
77       int8 in_data[2];     //Buffer para datos que serán recibidos.
78
79
80        usb_init();         //Inicializa módulo USB, espera por conexión esto si es buspowered y no hay
81                            //pin de sensado.
82       setup_adc_ports(AN0);
```

```
83
84          setup_adc(ADC_CLOCK_INTERNAL);
85          set_adc_channel(0);
86
87           while (TRUE) {
88               //usb_task(); //no hace falta si uso usb_init() en vez de usb_init_cs().
89             if (usb_enumerated()) {               //Pregunto si el dispositivo fue enumerado por PC para p
90                                                   //mandar/recibir paquetes de datos.
91                   out_data[0]=read_adc();         //Leo conversor AD y lo guardo.
92                   out_data[1]=!input(BUTTON);     //Leo estado del botón y lo guardo.
93                   usb_put_packet(1, out_data, 2, USB_DTS_TOGGLE);  //Envío valores AD y botón a PC.
94
95                if (usb_kbhit(1)) {                //Pregunto si el enpoint 1 tiene data en su buffer de sa
96                   usb_get_packet(1, in_data, 2);  //Lee 2 bytes del endpoint 1 y los guarda en in_data.
97                   output_b(in_data[0]);           //Cargo al puerto B la data llegada en el primer byte de
98                }
99                delay_ms(1);
100            }
101         }
102     }
```

## Anexo B. Descriptor USB del dispositivo

```
////////////////////////////////////////////////////////////////////////////
////                                                                    ////
////                        NI_usb_desc_hid.h                           ////
////                                                                    ////
//// Ejemplo de configuración de dispositivo "descriptor" para usar     ////
//// con el compilador CCS's (Adaptado del descriptor de un ejemplo     ////
//// del CCS, ver HID demo)                                             ////
////                                                                    ////
////////////////////////////////////////////////////////////////////////////

#IFNDEF __USB_DESCRIPTORS__
#DEFINE __USB_DESCRIPTORS__

#include <usb.h>

//////////////////////////////////////////////////////////////////////////
///
/// HID Report.  Tells HID driver how to handle and deal with
/// received data.  HID Reports can be extremely complex,
/// see HID specification for help on writing your own.
///
//////////////////////////////////////////////////////////////////////////

const char USB_CLASS_SPECIFIC_DESC[] = {
   6, 0, 255,     // Usage Page = Vendor Defined
   9, 1,          // Usage = IO device
   0xa1, 1,       // Collection = Application
   0x19, 1,       // Usage minimum
   0x29, 8,       // Usage maximum

   0x15, 0x80,       // Logical minimum (-128)
   0x25, 0x7F,       // Logical maximum (127)

   0x75, 8,       // Report size = 8 (bits)
   0x95, 2,       // Report count = 16 bits (2 bytes)
   0x81, 2,       // Input (Data, Var, Abs)
   0x19, 1,       // Usage minimum
   0x29, 8,       // Usage maximum
   0x75, 8,       // Report size = 8 (bits)
   0x95, 2,       // Report count = 16 bits (2 bytes)
   0x91, 2,       // Output (Data, Var, Abs)
   0xC0           // End Collection
};

//if a class has an extra descriptor not part of the config descriptor,
// this lookup table defines where to look for it in the const
// USB_CLASS_SPECIFIC_DESC[] array.
```

```
49      //first element is the config number (if your device has more than one config)
50      //second element is which interface number
51      //set element to 0xFFFF if this config/interface combo doesn't exist
52      const int16 USB_CLASS_SPECIFIC_DESC_LOOKUP[USB_NUM_CONFIGURATIONS][1] =
53      {
54      //config 1
55         //interface 0
56            0
57      };
58
59      //if a class has an extra descriptor not part of the config descriptor,
60      // this lookup table defines the size of that descriptor.
61      //first element is the config number (if your device has more than one config)
62      //second element is which interface number
63      //set element to 0xFFFF if this config/interface combo doesn't exist
64      const int16 USB_CLASS_SPECIFIC_DESC_LOOKUP_SIZE[USB_NUM_CONFIGURATIONS][1] =
65      {
66      //config 1
67         //interface 0
68            32
69      };
70
71
72   ////////////////////////////////////////////////////////////////////
73   ///
74   ///   start config descriptor
75   ///   right now we only support one configuration descriptor.
76   ///   the config, interface, class, and endpoint goes into this array.
77   ///
78   ////////////////////////////////////////////////////////////////////
79
80      #DEFINE USB_TOTAL_CONFIG_LEN      41  //config+interface+class+endpoint+endpoint (2 endpoints)
81
82      const char USB_CONFIG_DESC[] = {
83      //IN ORDER TO COMPLY WITH WINDOWS HOSTS, THE ORDER OF THIS ARRAY MUST BE:
84         //    config(s)
85         //    interface(s)
86         //    class(es)
87         //    endpoint(s)
88
89      //config_descriptor for config index 1
90            USB_DESC_CONFIG_LEN, //length of descriptor size          ==1
91            USB_DESC_CONFIG_TYPE, //constant CONFIGURATION (CONFIGURATION 0x02)     ==2
92            USB_TOTAL_CONFIG_LEN,0, //size of all data returned for this config      ==3,4
93            1, //number of interfaces this device supports       ==5
94            0x01, //identifier for this configuration.  (IF we had more than one configurations)      ==6
95            0x00, //index of string descriptor for this configuration      ==7
96            0xC0, //bit 6=1 if self powered, bit 5=1 if supports remote wakeup (we don't), bits 0-4 unused and bit7=1         ==8
```

```
97          0x32, //maximum bus power required (maximum milliamperes/2)  (0x32 = 100mA)
98
99      //interface descriptor 1
100         USB_DESC_INTERFACE_LEN, //length of descriptor      =10
101         USB_DESC_INTERFACE_TYPE, //constant INTERFACE (INTERFACE 0x04)       =11
102         0x00, //number defining this interface (IF we had more than one interface)    ==12
103         0x00, //alternate setting     ==13
104         2, //number of endpoins, except 0 (picl67xx has 3, but we dont have to use all).       ==14
105         0x03, //class code, 03 = HID     ==15
106         0x00, //subclass code //boot     ==16
107         0x00, //protocol code      ==17
108         0x00, //index of string descriptor for interface      ==18
109
110     //class descriptor 1  (HID)
111         USB_DESC_CLASS_LEN, //length of descriptor    ==19
112         USB_DESC_CLASS_TYPE, //dscriptor type (0x21 == HID)      ==20
113         0x00,0x01, //hid class release number (1.0) (try 1.10)      ==21,22
114         0x00, //localized country code (0 = none)       ==23
115         0x01, //number of hid class descrptors that follow (1)      ==24
116         0x22, //report descriptor type (0x22 == HID)                 ==25
117         USB_CLASS_SPECIFIC_DESC_LOOKUP_SIZE[0][0], 0x00, //length of report descriptor            ==26,27
118
119     //endpoint descriptor
120         USB_DESC_ENDPOINT_LEN, //length of descriptor                   ==28
121         USB_DESC_ENDPOINT_TYPE, //constant ENDPOINT (ENDPOINT 0x05)          ==29
122         0x81, //endpoint number and direction (0x81 = EP1 IN)       ==30
123         0x03, //transfer type supported (0x03 is interrupt)         ==31
124         USB_EP1_TX_SIZE,0x00, //maximum packet size supported                ==32,33
125         250,  //polling interval, in ms.  (cant be smaller than 10)      ==34
126
127     //endpoint descriptor
128         USB_DESC_ENDPOINT_LEN, //length of descriptor                   ==35
129         USB_DESC_ENDPOINT_TYPE, //constant ENDPOINT (ENDPOINT 0x05)          ==36
130         0x01, //endpoint number and direction (0x01 = EP1 OUT)      ==37
131         0x03, //transfer type supported (0x03 is interrupt)         ==38
132         USB_EP1_RX_SIZE,0x00, //maximum packet size supported                ==39,40
133         10 //polling interval, in ms.  (cant be smaller than 10)    ==41
134     };
135
136     //****** BEGIN CONFIG DESCRIPTOR LOOKUP TABLES ********
137     //since we can't make pointers to constants in certain picl6s, this is an offset table to find
138     //  a specific descriptor in the above table.
139
140     //NOTE: DO TO A LIMITATION OF THE CCS CODE, ALL HID INTERFACES MUST START AT 0 AND BE SEQUENTIAL
141     //      FOR EXAMPLE, IF YOU HAVE 2 HID INTERFACES THEY MUST BE INTERFACE 0 AND INTERFACE 1
142     #define USB_NUM_HID_INTERFACES   1
143
144     //the maximum number of interfaces seen on any config
```

```
145     //for example, if config 1 has 1 interface and config 2 has 2 interfaces you must define this as 2
146     #define USB_MAX_NUM_INTERFACES   1
147
148     //define how many interfaces there are per config.  [0] is the first config, etc.
149     const char USB_NUM_INTERFACES[USB_NUM_CONFIGURATIONS]={1};
150
151     //define where to find class descriptors
152     //first dimension is the config number
153     //second dimension specifies which interface
154     //last dimension specifies which class in this interface to get, but most will only have 1 class per interface
155     //if a class descriptor is not valid, set the value to 0xFFFF
156     const int16 USB_CLASS_DESCRIPTORS[USB_NUM_CONFIGURATIONS][1][1]=
157     {
158     //config 1
159        //interface 0
160           //class 1
161           18
162     };
163
164     #if (sizeof(USB_CONFIG_DESC) != USB_TOTAL_CONFIG_LEN)
165        #error USB_TOTAL_CONFIG_LEN not defined correctly
166     #endif
167
168
169  ////////////////////////////////////////////////////////////////////////
170  ///
171  ///   start device descriptors
172  ///
173  ////////////////////////////////////////////////////////////////////////
174
175     const char USB_DEVICE_DESC[USB_DESC_DEVICE_LEN] ={
176        //starts of with device configuration. only one possible
177           USB_DESC_DEVICE_LEN, //the length of this report   ==1
178           0x01, //the constant DEVICE (DEVICE 0x01)  ==2
179           0x10,0x01, //usb version in bcd (pic167xx is 1.1) ==3,4
180           0x00, //class code ==5
181           0x00, //subclass code ==6
182           0x00, //protocol code ==7
183           USB_MAX_EP0_PACKET_LENGTH, //max packet size for endpoint 0. (SLOW SPEED SPECIFIES 8) ==8
184           0xD8,0x04, //vendor id Identificador del fabricante y del Driver a usar !!!!
185           0x21,0x11, //product id   Identificador del Producto y del Driver a usar !!!!
186           0x14,0x13, //device release number  ==13,14
187           0x01, //index of string description of manufacturer. therefore we point to string_1 array (see below)  ==15
188           0x02, //index of string descriptor of the product  ==16
189           0x00, //index of string descriptor of serial number  ==17
190           USB_NUM_CONFIGURATIONS  //number of possible configurations  ==18
191     };
192
```

```
193
194     ////////////////////////////////////////////////////////////////////
195     ///
196     ///   start string descriptors
197     ///   String 0 is a special language string, and must be defined.
198     ///   People in U.S.A. can leave this alone.
199     ///
200     ///   You must define the length else get_next_string_character() will not see the string
201     ///   Current code only supports 10 strings (0 thru 9)
202     ///
203     ////////////////////////////////////////////////////////////////////
204
205     //the offset of the starting location of each string.
206     //offset[0] is the start of string 0, offset[1] is the start of string 1, etc.
207     char USB_STRING_DESC_OFFSET[]={0,4,12};
208
209     char const USB_STRING_DESC[]={
210        //string 0
211              4, //length of string index
212              USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
213              0x09,0x04,   //Microsoft Defined for US-English x4
214        //string 1
215              8, //length of string index
216              USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
217              'U',0,
218              'S',0,
219              'B',0,
220
221        //string 2
222              26, //length of string index
223              USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
224              'Y',0,
225              'O',0,
226              'H',0,
227              'A',0,
228              'N',0,
229              ' ',0,
230              ' ',0,
231              'T',0,
232              'A',0,
233              'R',0,
234              'E',0,
235              'A',0
236     };
237
238     #ENDIF
239
```