Universidad Nacional Autónoma de México

Facultad de Ingeniería

DESARROLLO DE UNA INTERFAZ USB PARA EL CONTROL REMOTO DE SISTEMAS ELECTROMECÁNICOS

TESIS

Para obtener el título de

INGENIERO EN COMPUTACIÓN

Presenta

ABRAHAM ISRAEL MONRROY CANO

Director de tesis

YUKIHIRO MINAMI KOYAMA

México, D.F. noviembre 2007

Contenido

Resumen

El objetivo del presente trabajo es desarrollar un sistema de comunicación para el control remoto de laboratorios para la realización de prácticas de las asignaturas de cinemática y dinámica, esto, para crear las versiones que puedan desarrollarse en forma remota de las que se imparten actualmente en el Laboratorio de Mecánica, facilitando el control a distancia de actuadores, motores, y equipos tanto electrónicos como electromecánicos que permiten la automatización de dichas prácticas. Además de implementar el sistema de comunicación, también se desarrollaron interfaces de los puertos que se encuentran en los equipos de cómputo, puerto serial, paralelo y Bus de Serie Universal (USB, por sus siglas en inglés), ofreciendo así una amplia gama de posibilidades a los desarrolladores de los equipos electrónicos para conectar sus versiones automatizadas de las prácticas al servidor que las controlará vía Internet.

Para el desarrollo del hardware, se utilizaron microcontroladores que facilitan el trabajo con los puertos, encargándose de los protocolos de comunicación y de la corrección y detección de errores en la transmisión de datos. El software de control remoto se implementó haciendo uso de la tecnología .NET de Microsoft, en forma de una página de Internet con contenido dinámico.

Los programas de comunicación y control local de los dispositivos de hardware (firmware) para los tres puertos se desarrollaron en lenguaje C para el microcontrolador que fue usado: el PIC18F4550 de Microchip, de alta velocidad.

El puerto USB requirió además del desarrollo de un controlador para Windows, para permitir la comunicación a altas velocidades entre el sistema operativo y el hardware externo.

Finalmente se integraron los dispositivos de hardware con el servicio de Internet para poder ser controlados remotamente.

Capítulo 1.

Introducción

Desde hace unas décadas, las comunicaciones se han convertido en una parte importante de nuestra sociedad, y de nuestra vida cotidiana, ya que nos permiten estar informados, comunicarnos con nuestros seres cercanos sin importar el lugar ni la hora, nos entretienen y nos facilitan algunas tareas, entre otras cosas.

El manejo a distancia de equipos, tanto electrónicos como mecánicos, permite la integración de servicios remotos, cuyo uso elimina la necesidad de personal especializado en sitio, disminuye los costos, facilita la administración, el mantenimiento y la automatización de los sistemas implementados, hace sencilla la expansión de proyectos, y además puede aprovechar las redes existentes, incluyendo el Internet.

La idea de desarrollar el sistema de control y comunicación, surgió debido a que la oferta de cupo en el Laboratorio de Mecánica de nuestra facultad, no es suficiente para atender a todos los alumnos que desean inscribirse a éste, por lo que nació la idea de ofrecer las prácticas vía Internet, permitiendo a los alumnos entrar y desarrollar sus prácticas tal y como si estuvieran allí, pero sin importar el día ni la hora y de esta forma cubrir la demanda existente.

El objetivo del presente trabajo es desarrollar un sistema de comunicación y control remoto de equipos electrónicos y electromecánicos tales como motores, actuadores, disparadores, cámaras digitales, convertidores analógicos digitales, digitales analógicos, sensores de presión y sonares, entre otros componentes que faciliten la automatización de tareas y la lectura precisa de datos, para permitir la realización remota de las prácticas de Cinemática y Dinámica que actualmente se imparten.

Este sistema debe ser eficaz, confiable y vanguardista, requerir mantenimiento mínimo, y ofrecer a la vez robustez, facilidad de uso y un costo accesible.

En el capítulo 2 se presentan los puertos que serán utilizados, así como los servicios remotos de control de equipo a través de Internet que se pretenden usar, una explicación de su funcionamiento, características principales, y las ventajas que ofrece su uso.

En el capítulo 3 se abordan las posibles soluciones para la creación de las interfaces, los dispositivos de hardware que se consiguen en el mercado y que pueden ser utilizados, y se fundamenta la selección de los componentes que se utilizarán, así como la implementación de los dispositivos de hardware diseñados, incluyendo el firmware generado para estos. Se muestra también el desarrollo del controlador para Windows que se requiere para la comunicación con el dispositivo USB, y también se presenta la creación del servicio de Internet para control remoto de las aplicaciones.

En el Capítulo 4 se presenta el desarrollo final del sistema, que integra tanto las interfaces de hardware como el servicio de Internet y su control, y el diseño final de las aplicaciones adaptadas al control remoto, para mejorar el desempeño y seguridad de éstas.

Capítulo 2

Puertos y servicios remotos

Los puertos de un equipo, también conocidos como periféricos, son aquéllos que permiten la comunicación con el mundo externo desde los programas de software, conectándolos con otros equipos electrónicos o mecánicos, por lo que son una parte fundamental en el desarrollo de este proyecto.

2.1 Puerto serial

La transmisión serial de datos usando el estándar RS-232, ha existido desde 1969 cuando se usaban teletipos y módems electromecánicos. Cuando se diseño este estándar, nunca se pensó que se llegaría a usar en equipos electrónicos tales como PCs, cámaras, teléfonos móviles, etc., por lo que pasó por muchas modificaciones y revisiones, la última es la revisión *TIA-232-F* emitida en 1997.

En esta investigación *sólo se analizará el estándar TIA-574* que define las transmisiones seriales apegándose a TIA-232-F, de manera *asíncrona* usando el conector serial *DE-9*, ya que este tipo de conexión es el implementado en las computadoras personales.

Como se mencionó, el estándar al ser diseñado ya hace bastantes años, sólo permite la comunicación entre dos equipos, no define codificación de caracteres, tampoco protocolos de detección de errores, algoritmos de compresión, ni velocidades de transmisión, sólo define que soporta velocidades menores a 20,000 bits por segundo, aunque la mayoría de los dispositivos actuales pueden superar esta velocidad, comúnmente se encuentran dispositivos que soportan velocidades de 38,400 ó 57,600 bits por segundo, y en algunas ocasiones hasta 115,200 e incluso 230,400 bits por segundo.

A continuación se muestran algunos detalles mecánicos, eléctricos y funcionales definidos en el estándar.

2.1.1 Conectores

Como se mencionó, el estándar TIA-574 define el uso del conector DE-9, cuya asignación definida de pines es la mostrada en la Figura 1 y que son descritas en la Tabla 1:

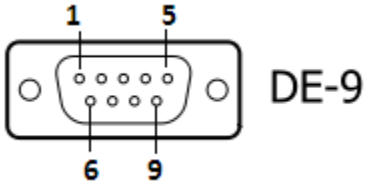


Figura 1. Asignación de pines del conector DE-9

Pin	Nombre	Descripción de la Señal
1	CD Carrier Detect	Detección de Portador. Es usado para detectar si hay otro dispositivo conectado
2	RXD Receive Data	Recepción de Datos. Se reciben los datos transmitidos por el otro dispositivo
3	TXD Transmit Data	Transmisión de Datos. Se envían los datos a otro dispositivo
4	DTR Data Terminal Ready	Datos listos en Terminal. Usado para indicar que está listo para transmitir
5	GND Signal Ground	Tierra común
6	DSR Data Set Ready	Datos listos para enviar. Indica que el otro dispositivo está listo para transmitir
7	RTS Request to Send	Petición para Envío. Para indicar que se está listo para enviar
8	CTS Clear to Send	Limpiar para Enviar. Para indicar que el otro dispositivo está listo para recibir
9	RI Ring indicator	En modem serial indica que se tiene una llamada

Tabla 1. Descripción de las señales en el conector DE-9

2.1.2 Códigos de línea y voltajes

La transmisión serial basada en el estándar RS-232 utiliza señales desbalanceadas, unidireccionales, y tienen una tierra común. Las señales están codificadas en NRZ[1] y son asíncronas. Normalmente los datos son transmitidos en palabras de 7 u 8 bits. Un bit de Inicio, marca el comienzo de una palabra. Este bit de inicio es un cero lógico. En la figura 2 se muestra una palabra de 8 bits, antes de ser invertida, encerrada entre los bits de inicio y parada.

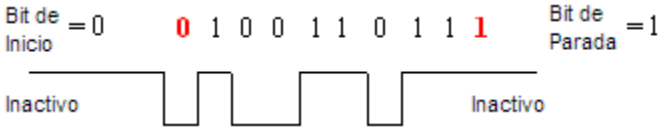


Figura 2. Codificación NRZ para un byte con sus respectivos bits de inicio y parada

[1] En telecomunicaciones, un código de línea NRZ (*Non Return to Zero*) es un código binario en que los unos lógicos son representados por una condición significativa, y los ceros lógicos son representados por otra condición significativa, sin ninguna otra condición neutral.

Las convenciones utilizadas por las señales, para indicar el estado lógico en términos de voltajes son las siguientes:

Rango de voltaje	Nivel lógico
+3 a +15 V	0
-3 a -15 V	1

Tabla 2. Codificación de niveles lógicos en RS-232

La región de voltaje comprendida entre -3 V y +3 V, se le considera región de transición, donde el estado de la señal es indefinido.

El estándar también define que el voltaje máximo en circuito abierto deberá ser de 25 V.

2.1.3 Transmisión serial asíncrona

Cada puerto COM (**Com**unicaciones) en una computadora personal es un puerto serial asíncrono controlado por un UART (*Universal Asynchronous Receiver Transmitter*, Transmisor Receptor Universal Asíncrono) y convencionalmente tiene una interfaz RS-232.

Al decir que la transmisión de los datos es serial, nos referimos a que se envía un bit a la vez en forma secuencial; cuando se dice que es asíncrono quiere decir que en la transmisión no se incluye una señal de reloj que sincronice el envío y la recepción de datos, sino que tanto el transmisor como el receptor tienen su propio reloj, oscilando a una frecuencia previamente establecida por ambos. Un UART es un dispositivo que hace la conversión entre transmisiones seriales y paralelas[2], y viceversa, en otras palabras, es capaz de enviar y recibir flujos de bits en una transmisión serial, y obtener su equivalente en representación paralela o de byte (8 bits simultáneos).

2.1.4 Transmisión serial con tres cables

Normalmente la transmisión serial hacia un microcontrolador o micro procesador no requiere todos los cables definidos en el estándar, sino que se usa una versión en la que sólo se requieren 3 líneas de comunicación, a saber TXD, RXD y GND, y el resto de las líneas son omitidas. A este método de comunicación se le denomina flujo de control por software, o también XON/XOFF, es decir sólo se controla la transmisión enviando un bit de inicio y otro de parada; esta forma de comunicación aún es ampliamente usada en los equipos electrónicos actuales tales como teléfonos móviles, cámaras digitales, computadoras de mano, etc.

2.1.5 Conversión de señales RS-232 a TTL

La mayoría de los microcontroladores y microprocesadores en el mercado trabajan con voltajes compatibles con TTL, donde un uno lógico es determinado por 5 V y un cero lógico es determinado por 0 V. Como se señaló anteriormente las señales RS-232 pueden variar desde los -15 V hasta los 15 V, por lo

[2] AXELSON, Jan. *"Serial Port Complete, Programming and Circuits for RS-232 and RS-485 Links and Networks"*, p. 26.

que son incompatibles y la inserción de una señal RS-232 a un dispositivo compatible con TTL lo dañaría dejándolo inservible, por lo que es necesario hacer una conversión entre este tipo de señales. Para lograrlo existen circuitos integrados que hacen esta función, llamados controladores y receptores RS-232.

2.1.6 Transferencia de datos con el puerto serial

Un método para poder enviar datos por el puerto serial requiere saber cuál es la dirección base del puerto. Las direcciones de los puertos seriales siempre se ordenan en el sistema básico de entrada y salida (BIOS, por sus siglas en inglés) de la siguiente manera: 3F8h, 2F8h, 3E8h y 2E8h[3](la h delante de los números indica que son representaciones de números hexadecimales), estas direcciones se encuentran almacenados como datos en la dirección de memoria 0040:0000h[4]. Generalmente se llegan a encontrar hasta dos puertos inherentes al equipo, pero es mucho más común encontrar sólo un puerto (en los equipos más modernos y económicos ya no se incluye ningún puerto serial). Ahora que ya se conoce la dirección base sólo se escriben los datos en forma de byte para que estos sean transmitidos por la línea.

Como se comentó, es posible configurar la transmisión, su velocidad, bits de datos, y un elemento llamado bit de paridad que es opcional, fue agregado al estándar debido a que no se cuenta con algún método para detectar errores de transmisión. Para hacer la configuración, sólo se debe escribir al puerto en registros especiales de configuración[5].

Para el establecimiento de velocidad de transmisión, se utiliza la dirección base leída en el BIOS para el puerto por el que se desea transmitir y se le suma 00h
La siguiente Tabla 3 muestra los valores que pueden ser asignados a esta dirección para la configuración de este parámetro.

Valor a escribir en el puerto	Velocidad (bits por segundo)
0x03 (predeterminado)	38,400
0x01	115,200
0x02	57,600
0x06	19,200
0x0C	9,600
0x18	4,800
0x30	2,400

Tabla 3. Valores de configuración para establecimiento de velocidad de transmisión

[3] AXELSON, Jan. *"Serial Port Complete, Programming and Circuits for RS-232 and RS-485 Links and Networks".* p. 29.
[4] En equipos basados en arquitectura Intel, la memoria es organizada por páginas y para navegar dentro de la página se utiliza un número denominado *Offset,* para representar lo anterior se utiliza la notación XXXX:YYYYh, donde XXXX representa el número de página y YYYY representa el offset dentro de la página.
[5] Serial port pins and registers www.captain.at/serial-port-registers.php.

Para el establecimiento de los bits de datos, bit de paridad, y bits de parada, deberá escribirse el valor correspondiente según la Tabla 4 en la dirección base del puerto, leída en el BIOS, sumándole 03h

bit #	Descripción			
0 & 1	B1	B0		tamaño de la palabra
	0	0		5 bits
	0	1		6 bits
	1	0		7 bits
	1	1		8 bits
2	0			un bit de parada
	1			2 bits de parada para palabras de tamaño de 6,7 y 8 bits. Ó 1.5 bits de parada para palabras de 5 bits.
3, 4 & 5	B5	B4	B3	
	X	X	0	sin paridad
	0	0	1	paridad impar
	0	1	1	paridad par
	1	0	1	paridad alta (bit siempre alto)
	1	1	1	paridad baja (bit siempre bajo)
6				habilitar aborto de transmisión
7	1			acceso al divisor
	0			acceso al buffer de recepción, de transmisión y al registro de habilitación de interrupciones.

Tabla 4. Byte de configuración para establecimiento de tamaño de palabra, paridad y bits de parada

Lo más comúnmente utilizado en transmisión serial son 8 bits de datos, 1 bit de parada, sin bit de paridad. Esto sería la palabra de configuración 03h, para este registro.

2.2 Puerto paralelo

Como su nombre lo indica, el puerto paralelo, a diferencia del puerto serial, obtiene los datos de manera paralela, es decir, simultánea, los bits incluidos en una palabra son transmitidos en un mismo instante.

Esta interfaz fue desarrollada por Centronics en los Laboratorios Wang, originalmente usaba un conector Amphenol de 36 pines y fue utilizado para una de sus primeras calculadoras. Más tarde cuando IBM implementó el puerto paralelo en la computadora personal en 1981, lo hizo usando el conector DB-25F. Después Hewlett Packard utilizó el puerto Centronics paralelo para incluirlo en sus impresoras e introdujo una versión bidireccional que denominó *Bitronics* en 1992[6].

En 1994 las interfaces Bitronics y Centronics fueron reemplazadas por el estándar IEEE 1284.

[6] Warp Nine Engineering – Parallel Port Background. www.fapo.com/porthist.htm.

2.2.1 Conectores

El estándar IEEE 1284 define el uso de tres distintos conectores:

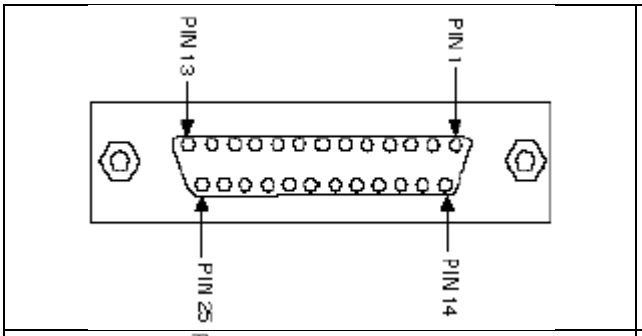
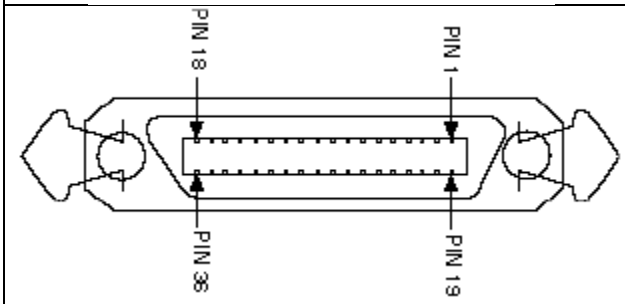
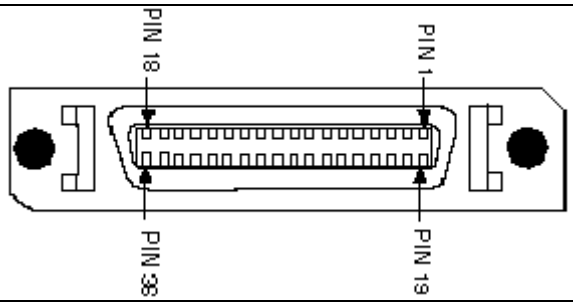
	DB-25 para la conexión al servidor *conector tipo A*
	Centronics, de 36 pines, para la conexión a la impresora o equipo terminal. *conector tipo B*
	Mini-Centronics, de 36 pines, un conector más pequeño que nunca se hizo popular. *conector tipo C*

Tabla 5.Conectores utilizados en el Puerto IEEE 1284

En la Tabla 6 se muestra la asignación de los pines de cada conector.

Pin	Descripción de los pines		
	Conector A	*Conector B*	*Conector C*
1	STROBE*	STROBE*	BUSY
2	Data1	Data1	Select
3	Data2	Data2	ACK*
4	Data3	Data3	FAULT*
5	Data4	Data4	PError
6	Data5	Data5	Data1
7	Data6	Data6	Data2
8	Data7	Data7	Data3
9	Data8	Data8	Data4
10	ACK*	ACK*	Data5
11	BUSY	BUSY	Data6
12	PError	PError	Data7
13	Select	Select	Data8

14	AUTOFD*	AUTOFD*	INIT*
15	FAULT*	No definido	STROBE*
16	INIT*	Logic Ground	Select In*
17	Select In*	Chassis Ground	AUTOFD*
18	Tierra	Peripheral Logic High	Host Logic High
19	Tierra	Tierra	Tierra
20	Tierra	Tierra	Tierra
21	Tierra	Tierra	Tierra
22	Tierra	Tierra	Tierra
23	Tierra	Tierra	Tierra
24	Tierra	Tierra	Tierra
25	Tierra	Tierra	Tierra
26		Tierra	Tierra
28		Tierra	Tierra
29		Tierra	Tierra
30		Tierra	Tierra
31		INIT*	Tierra
32		FAULT*	Tierra
33		No definido	Tierra
34		No definido	Tierra
35		No definido	Tierra
36		Select In*	Peripheral Logic High

Tabla 6. Asignación de pines en los conectores para el puerto IEEE 1284

El asterisco mostrado delante de algunas señales, indica que estas líneas funcionan con lógica inversa

Como puede observarse, el puerto cuenta con 17 líneas para señales, y 8 para tierra. Las líneas de señales podemos clasificarlas en tres tipos, a continuación se muestra una breve descripción de las líneas tal y como fueron diseñadas para el uso en una impresora.

Control (4 líneas):

Strobe	usado para indicar cuando los datos han sido escritos y se encuentran estables
AUTOFD	auto alimentación de papel
INIT	inicializar impresora
Select In	para indicar que se utilizará la impresora conectada a ese puerto

Status (5 líneas):

ACK	usado para indicar recepción exitosa de datos
BUSY	indica que el dispositivo está ocupado para recibir datos
PERROR	para reportar error con el papel
Select	para indicar si la impresora está en línea
FAULT	error en la impresora

Datos (8 líneas):

Data0-Data7	líneas de datos, para transmitir información hacia la impresora

Las líneas de control son utilizadas para el control de la interfaz y para el establecimiento de la comunicación entre la PC y el dispositivo. Las líneas de Status son usadas para establecer la comunicación entre la impresora y la PC, para reportar el estado de la impresora, así como para indicar

errores. Las líneas de datos son para enviar información sólo desde la PC hacia el periférico. Posteriores implementaciones permitieron la comunicación en ambos sentidos por estas líneas.

2.2.2 Modos de transferencia.

La Tabla 7 muestra los modos de transferencia definidos en la especificación liberada en 1994.

Modo de Transferencia	Velocidad Máxima (B/s)
Compatible (Centronics)/SPP	360,360
Nibble	3,174,603
Byte	1,369,863
EPP	3,333,333
ECP	2,500,000

Tabla 7. Velocidades de los modos de transferencia para el puerto IEEE 1284

Estos son los modos existentes y definidos en IEEE 1284; en este trabajo se hablará sólo del modo compatible, también llamado SPP, ya que es el que es soportado por los microcontroladores y microprocesadores, de manera inherente, ya que no requiere de la implementación de los protocolos definidos por el estándar; los otros modos requieren un control especial, generalmente diseñado para el uso con impresoras que se encargan de llevar a cabo las funciones definidas en el protocolo de comunicación.

2.2.3 Transferencia con modo compatible SPP

El puerto paralelo para poder ser accedido por software, se le asigna una dirección tal y como si fuera una memoria, donde sólo se direcciona y el valor requerido es leído por las líneas de datos. Antes de conocer los registros del puerto, es necesario conocer cuál es la dirección base del puerto que se tiene instalado en la computadora; generalmente al puerto paralelo se le asigna la dirección 378h, pero debido a que esto puede ser modificado desde el Sistema Básico de Entrada y Salida de la PC, es necesario verificar este dato, por lo que para poder saber cuál es la dirección base del primer puerto paralelo en la PC (Si es que hay más de uno), es necesario acceder a la dirección 0040:0008h, si la PC contara con más puertos paralelos, las siguientes direcciones base de los puertos estarían en la dirección siguiente, es decir 0040:000Ah, ya que son registros de dos bytes, y así sucesivamente si se tuvieran más puertos.

Ahora que ya se conoce la dirección base, se describirán los registros a los que se puede acceder.

El registro de datos o puerto de datos (dirección base + 00h), es usado como salida de los datos en las líneas de datos del puerto (pines 2-9). Este puerto es sólo de salida, pero en los otros modos, puede ser usado bidireccionalmente. Si se lee del puerto como entrada, se obtendrá el último valor que se haya escrito.

Bit #	7	6	5	4	3	2	1	0
Descripción	Data7	Data6	Data5	Data4	Data3	Data2	Data1	Data0

El registro de status o puerto de status (dirección base +01h), es un puerto de sólo lectura. Cualquier intento de escritura será ignorado. Este puerto tiene 5 líneas (pines 10, 11, 12, 13 y 15), un registro IRQ[7] y dos bits reservados. Es de notarse que tanto el bit 7 como el 2, son entradas con lógica invertida.

Bit #	7	6	5	4	3	2	1	0
Descripción	Busy	Acknowledge	Paper out	Select In	Error	IRQ	Reservado	Reservado

El registro de control (dirección base + 02h) fue diseñado como de sólo escritura. Cuando una impresora es conectada al puerto, cuatro líneas de control son requeridas*: Strobe, Auto linefeed, Initialize y Select printer*, de los cuales sólo *Initialize* no es de lógica invertida.

Bit #	7	6	5	4	3	2	1	0
Descripción	No usado	No usado	Habilitar pto. bidireccional	Habilitar IRQ vía ACK	Select printer	Initialize (Reset)	Auto linefeed	Strobe

En este registro es necesario enfatizar que estás salidas también pueden ser utilizadas como entradas, ya que estas cuatro líneas son salidas de colector abierto.

2.3 Puerto USB, Bus de Serie Universal

Es una conexión serial para crear interfaces entre dispositivos y fue diseñado para crear una computadora personal libre de puertos considerados obsoletos (puertos serial RS-232 y paralelo IEEE 1284, analizados anteriormente). El principal objetivo de este bus fue el de conectar dispositivos de distintos tipos en un mismo puerto, además de mejorar las capacidades de *Plug-and-play*[8], para facilitar la detección de nuevos dispositivos y permitir conectar y desconectar dispositivos sin requerir reiniciar la computadora, e incluso crear clases de hardware que no requieran controladores para que el sistema operativo pueda comunicarse con estos; otro aspecto importante es el de alimentar al dispositivo por el mismo bus, al que se le pueden requerir hasta 500 [mA] que son suficientes para la mayoría de las aplicaciones en electrónica.

Un sistema USB tiene un diseño asimétrico, es decir, consiste de un *Host* y muchos dispositivos periféricos conectados a este en una forma denominada *Daisy-Chained*, es decir, es posible agregar equipos denominados *Hub* que permiten hacer crecer la cantidad de dispositivos conectados, creando una red tipo árbol, es decir con una topología tipo estrella, limitada hasta cinco subniveles por

[7] *Interrupt ReQuest*, Petición de Interrupción, es una línea especial que va del dispositivo hacia el microprocesador para requerir una interrupción, es decir para requerir atención del microprocesador, para indicar que hay información disponible para el envío hacia la PC.
[8] *Plug and Play*, Conecta y usa, es una tecnología diseñada para poder conectar y desconectar dispositivos periféricos sin necesidad de apagar o reiniciar el equipo, además de tener la capacidad de identificar los controladores del dispositivo para el sistema operativo de manera automática.

controlador. Este método de conexión permite la conexión de hasta 127 dispositivos a un solo controlador Host. En la Figura 3 se ilustra la topología descrita.

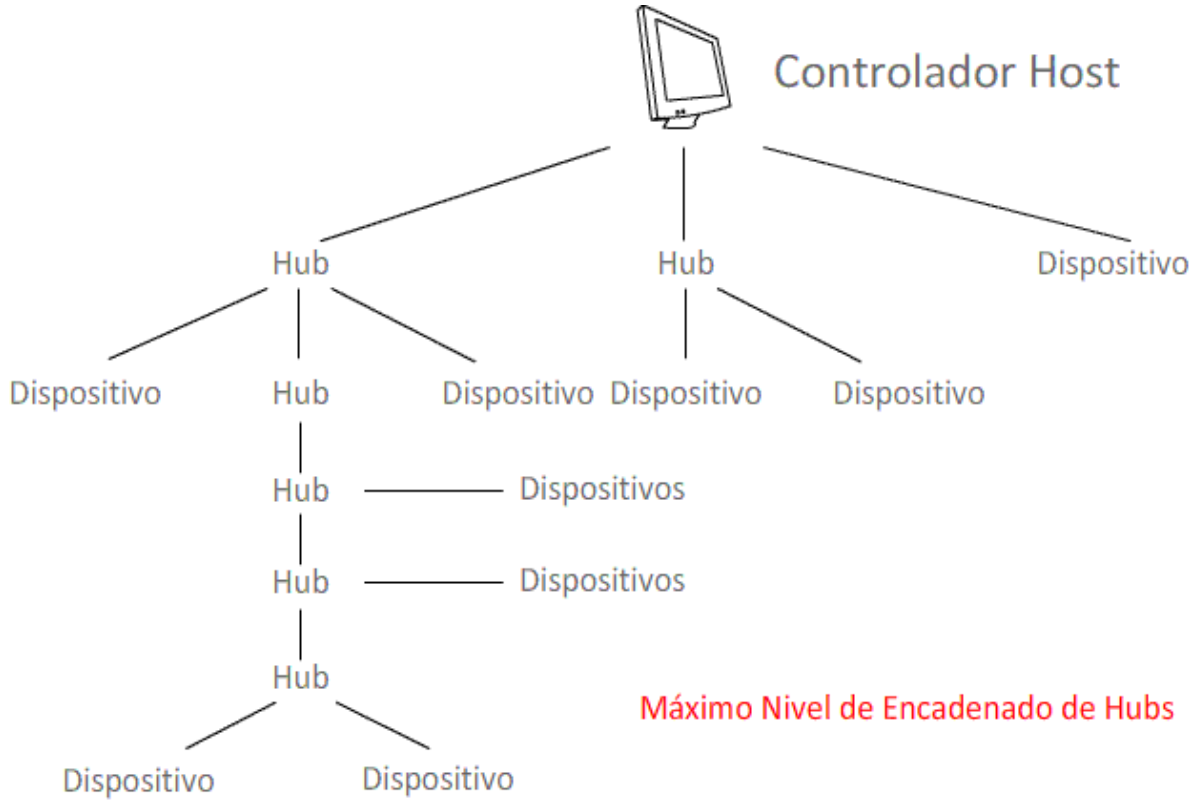


Figura 3. Diagrama que muestra la topología y el máximo nivel de encadenado en USB

El diseño del estándar USB es controlado por el USB-IF (USB Implementers Forum), que es una organización sin fines de lucro que promueve y da soporte al estándar USB. Los miembros más notables del foro son Apple, Hewlett-Packard, NEC, Microsoft, Intel y Agere.

A pesar de que esta tecnología fue desarrollada a finales de 1994, no fue sino hasta enero de 1996 cuando fue liberada la versión 1.0, que especificaba velocidades de 1.5 Mbps para dispositivos "*Low Speed*", y una velocidad de hasta 12 Mbps para los dispositivos "*Full Speed*". Pero esta primera versión tuvo muchos problemas, por lo que en Septiembre de 1998, se liberó la versión 1.1, que corregía todos los problemas encontrados y además agregaba otro modo de transferencia, denominado interrupción. En abril del año 2000, se liberó la versión 2.0, que agregaba una tercera velocidad para los dispositivos "High Speed", de hasta 480 Mbps que permitió el desarrollo de dispositivos de alto rendimiento.

2.3.1 Dispositivos USB de entrada/salida

Los dispositivos USB son elementos que generan o consumen datos desde el mundo exterior, o pueden hacer ambas funciones. Una conexión lógica o por software para un dispositivo USB se muestra en la Figura 4[9], es general y puede ser aplicada a cualquier dispositivo.

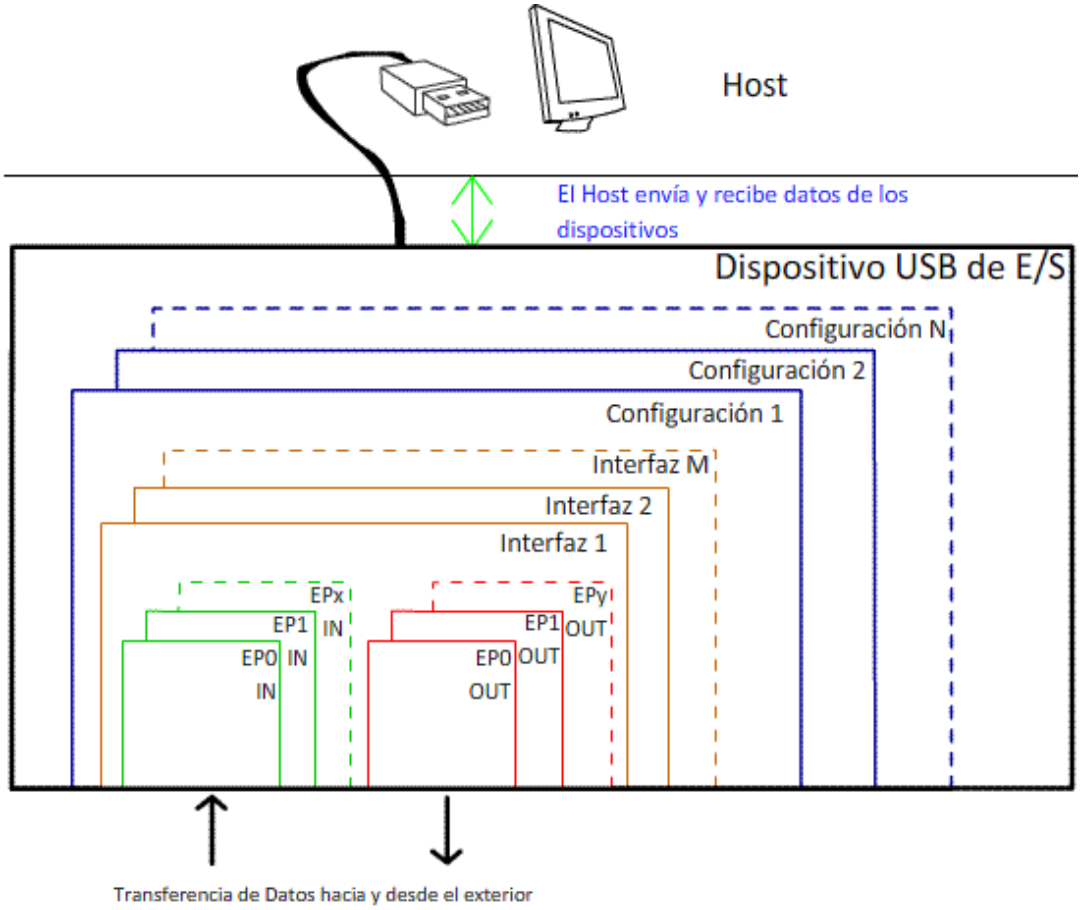


Figura 4. Esquema lógico de un dispositivo USB de Entrada/Salida

Las siglas EP mostradas en la Figura 4, son denominadas *Endpoints*, este término es usado para describir dónde los datos entran y salen el sistema USB. Un *Endpoint IN* es un generador de datos, mientras que un *Endpoint OUT* es un consumidor de datos. Un dispositivo común puede requerir varios endpoints para poder crear un esquema eficiente de transferencia de datos. A la colección de endpoints se le denomina **Interfaz** y está directamente relacionada con la conexión física. A una colección de interfaces se le denomina **Configuración**. Una configuración define los atributos y características de un dispositivo específico.

[9] Figura adaptada de *"USB Design by Example"* p. 13, por Hyde, John.

2.3.2 Cables y conectores

Un cable USB contiene dos cables de alimentación y dos de datos (D+ y D-) que deben de ir trenzados, como se muestra en la Figura 5. La distancia máxima para los cables es de cinco metros.

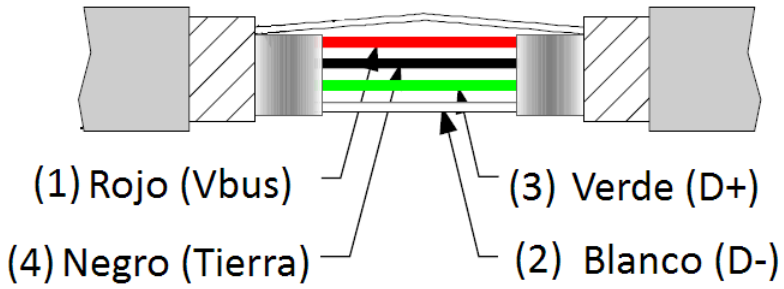
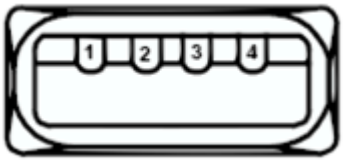
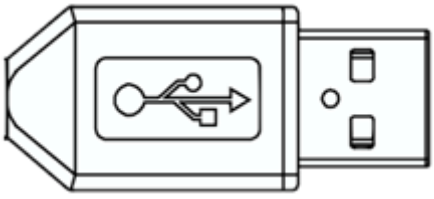
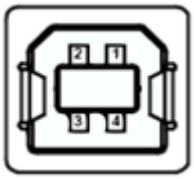
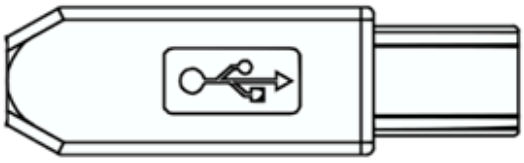
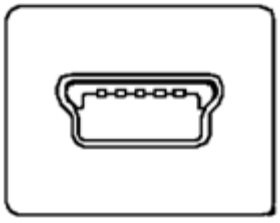
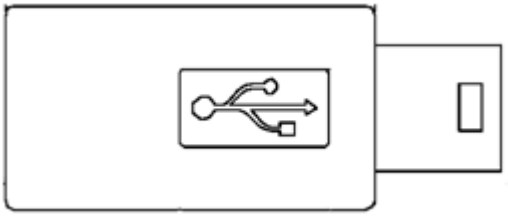


Figura 5. Líneas contenidas en un cable USB

La alimentación por bus es un importante beneficio de la especificación USB, de esta manera un dispositivo puede funcionar sólo con el cable del bus y así eliminar la fuente de poder externa, que acompaña a la mayoría de los dispositivos. Esta autoalimentación es controlada cuidadosamente por el *Host*, garantizando un voltaje mínimo de 4.75 V y un máximo de 5.25 V, donde es posible obtener hasta 500 mA por puerto.

Hay distintos conectores USB, que facilitan el desarrollo de distintos dispositivos con variedad de tamaños y características. La Tabla 8 muestra los conectores disponibles.

Tipo de plug	Vista frontal	Vista horizontal
Tipo A Usado en conexión hacia el Host		
Tipo B Usado en conexión hacia el dispositivo		
Tipo Mini-B Usado siempre hacia el dispositivo		

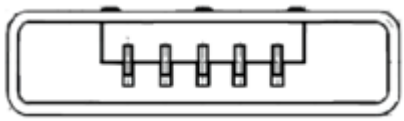
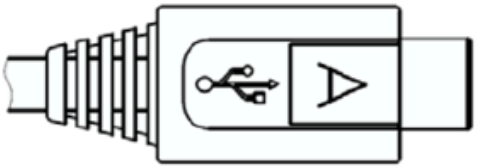
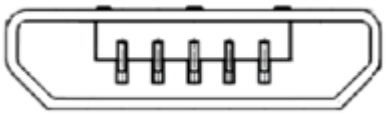
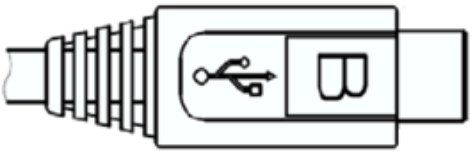
Tipo Micro A Usado siempre hacia Hub's		
Tipo Micro B Usado siempre hacia el dispositivo		

Tabla 8. Tipos de Conectores USB disponibles

2.3.3 Señalización

El estándar USB define el uso de señales diferenciales en la transmisión para reducir los efectos del ruido inducido por el sistema, estas se encuentran codificadas usando NRZI (*Non Return to Zero Inverted*), utilizando *bit stuffing* previamente. El código de línea *NRZI*, define que cuando se encuentra un "1" se representa por ningún cambio en el nivel lógico, y cuando se encuentra un "0" se hace un cambio de nivel lógico.

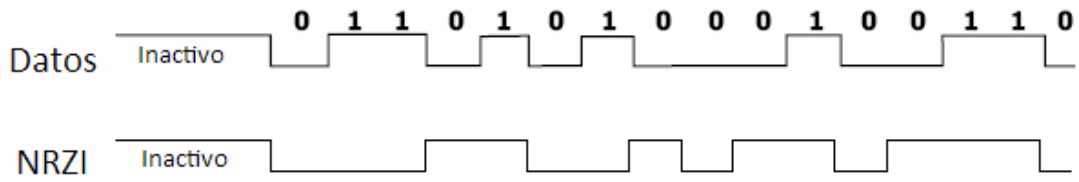


Figura 6. Ejemplo de codificación de datos en NRZI

El *bit stuffing* es una técnica utilizada para asegurar la correcta transición de la señal. Esta técnica es empleada por el transmisor de los datos cuando se envía un paquete USB. Un cero es insertado intencionalmente en la cadena después de seis unos consecutivos, antes de codificar en NRZI[10], esto para forzar una transición en la transmisión NRZI, y así evitar que el transmisor pierda sincronía de reloj, y comience a perder datos en la transmisión. El receptor deberá ser capaz de detectar estos bits insertados y eliminarlos.

Ahora para ejemplificar el uso de *bit stuffing* para luego codificar en NRZI, en la Figura 7 se muestra cómo se forma una parte del paquete USB, que posteriormente será detallado.

[10] USB-IF. *"Universal Serial Bus Revision 2.0 specification",* p.157.

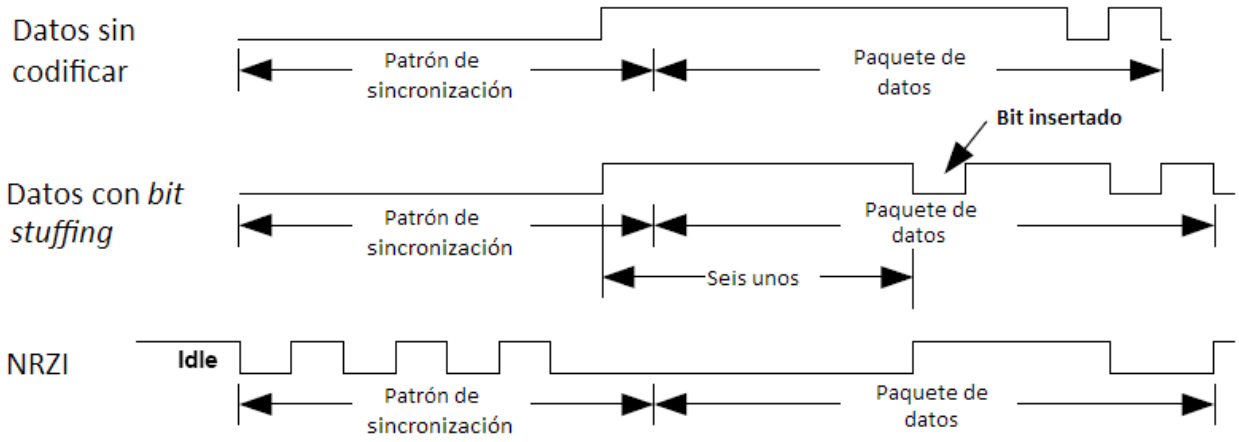


Figura 7. Codificación de un paquete USB para ser enviado por la línea de transmisión.

2.3.4 El Paquete básico USB

El bus USB es una conexión punto a punto que opera en modo *Half-Duplex*, es decir puede haber comunicación en el canal en ambos sentidos, pero no puede ser simultánea. El elemento fundamental de la comunicación es el **paquete**, y la secuencia definida de paquetes es usada para la construcción de un canal de comunicación robusto.

Un paquete consiste básicamente de tres piezas: un inicio, datos y un fin. En la Figura 8 se muestran las señales en forma diferencial, tal y como se observaría a través de un osciloscopio. Es de notarse que la información que a continuación se presenta **sólo** es aplicable a modos *Low* y *Full Speed*, para el modo *High-Speed*, las características del paquete son distintas, y no serán analizadas en este trabajo ya que la aplicación a realizar será en modo *Full Speed*.

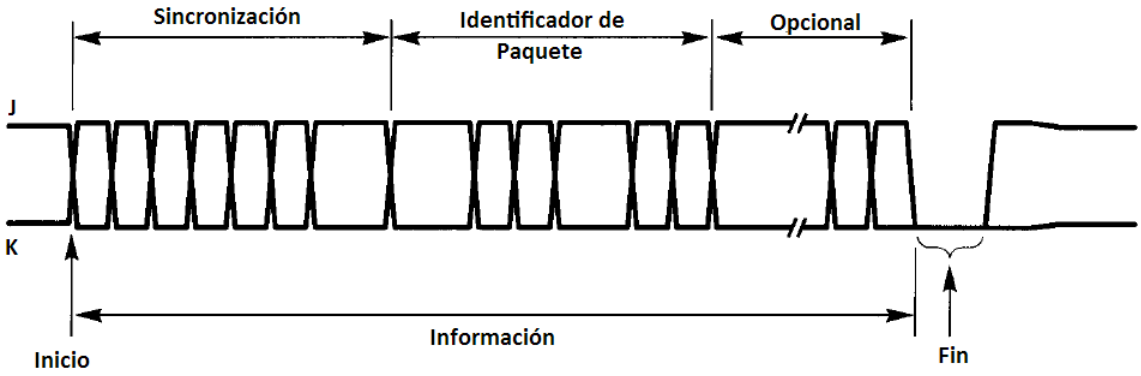


Figura 8. Señal diferencial USB

La información de un paquete puede ser desde 1 byte hasta 1025 bytes. El primer byte es siempre un identificador de paquete, o PID, que define cómo debe ser interpretada la información contenida. El PID está formado por 4 bits y los otros 4 son el complemento de los otros, esto le permite al receptor hacer

una verificación de errores al PID. De los 16 posibles identificadores permitidos por este esquema, diez están definidos y los restantes son reservados, como se muestra en la Tabla 9. Los números del valor del PID son binarios.

Valor de PID	Tipo de paquete	Categoría
0101	SOF	Token
1101	SETUP	Token
1001	IN	Token
0001	OUT	Token
0011	DATA0	Datos
1011	DATA1	Datos
0010	ACK	Handshake
1010	NAK	Handshake
1110	STALL	Handshake
1100	PRE	Especial
El Resto	Reservado	Reservado

Tabla 9. Equivalencia de valores del identificador de paquete

Los paquetes de categoría *Token,* son utilizados para configurar las transmisiones de los datos, que a su vez son reconocidos por los de clase *Handshake*, que son aquellos que se utilizan para el establecimiento de comunicación. También existen los paquetes *Especiales* que son utilizados en las comunicaciones de baja velocidad.

2.3.4.1 Paquete Token SOF

El *Host* transmite cada milisegundo un paquete *SOF* (para modos *Low* y *Full Speed*; para *High Speed* es de 125 µs). El tiempo entre dos paquetes SOF se le llama *Frame*. Un paquete SOF tiene 11 bits de datos y 5 bits de CRC[11]. Si el CRC calculado nuevamente por el receptor y no es igual, el paquete se descarta y no se realiza acción alguna.

Los 11 bits de datos, son solamente un contador que se incrementa monótonamente, por lo que cada 2048 ms vuelve a iniciar el contador. Este valor puede ser utilizado por dispositivos que trabajan en tiempo real para sincronizar sus datos. Este paquete está siempre presente en el bus, así que cualquier dispositivo conectado al bus, recibirá este paquete. Este paquete es el único que no tiene dirección de destino y que no requiere de ningún *acknowledge*, es decir una respuesta por parte del receptor para indicar que el paquete fue recibido correctamente.

[11] CRC (*Cyclic Redundancy Check*), es un método de detección de errores en la transmisión de datos denominado código de redundancia cíclica.

En la Tabla 10 se muestra cómo está compuesto el paquete SOF:

SOF	Número de frame	CRC
8 bits (0101 + 1010)	11 bits	5 bits

Tabla 10. Estructura del Paquete Start of frame

2.3.4.2 Paquetes Token SETUP, IN, OUT

Estos tres tipos de paquetes son usados para configurar la transmisión de datos entre el *Host* y una fuente de datos o un consumidor de datos (llamados *Endpoints*, como se verá más adelante) en el dispositivo.

Un paquete *IN*, configura la transmisión de datos desde el dispositivo hacia el *Host*, mientras que el paquete *OUT* lo hace desde el *Host* hacia el dispositivo.

Los paquetes *IN* y *OUT* pueden direccionar cualquier *Endpoint* en cualquier dispositivo.

Un paquete de tipo *SETUP* es un caso especial de un paquete *OUT*, es de "Alta prioridad", lo que implica que el dispositivo debe aceptar el paquete incluso si debe abortar la acción que estaba realizando. Y siempre se envía al Endpoint 0 (llamado de control y que es bidireccional, como se verá adelante).

En la Tabla 11 se muestra la composición de los paquetes *Token*.

SETUP / IN / OUT	Dirección de Dispositivo	Endpoint	CRC
8 Bits PID	7 bits	4 bits	5 bits

Tabla 11. Estructura de un paquete *Token*

2.3.4.3 Paquetes de datos

Las trasferencias iniciadas por paquetes *Token* SETUP, IN y OUT, son implementadas mediante los paquetes DATA0 y DATA1. Un paquete de transferencia de datos puede tener un peso desde 0 hasta 1023 bytes y un CRC de 16 bits (hay algunas restricciones para este número según el tipo de transferencia, como se analizará posteriormente). Existen dos tipos de paquetes DATA0 y DATA1 para proveer una verificación extra para la detección de errores, la cual consiste en que debe transmitirse un paquete de datos alternadamente, es decir un paquete DATA0 tras un DATA1 y nuevamente un DATA0. De esta manera si se reciben dos paquetes de tipo DATA0 o DATA1 de manera consecutiva, se tendrá un error de recepción, ya que se perdió un paquete. Su composición se muestra en la Tabla 12.

DATA0 / DATA1 PID	Datos	CRC
8 bits	De 0 a 1023 bytes	16 bits

Tabla 12. Estructura de un paquete de datos

2.3.4.4 Paquetes de handshake

Estos paquetes son usados por el receptor para indicar al transmisor el estado de la recepción de los paquetes *Token* o de datos. En estos paquetes no se provee CRC, sólo se cuenta con la detección de error por complemento en el PID.

El *ACK* indica la recepción exitosa de un paquete *Token* o de datos.

El *NAK* indica que actualmente el dispositivo se encuentra demasiado ocupado o no tiene los recursos necesarios para poder recibir el paquete de datos o *Token* en ese instante. Una nota importante es que el *Host* no puede enviar *NAK* a ninguna transacción, mientras que un dispositivo tiene permitido enviar *NAK* a cualquier transacción, excepto cuando el paquete *Token SETUP* es recibido. El *Host* siempre estará en una PC, por lo que siempre tendrá los recursos necesarios para poder recibir el paquete.

Si algo sucede mal en el dispositivo, entonces se generará el paquete de *handshake STALL* que le indica al *Host* que requiere de atención. Por ejemplo, un dispositivo generará un paquete *STALL* a todos los comandos que reciba y que no reconozca.

En la Tabla 13 se muestra la composición de este tipo de paquetes.

ACK / NAK / STALL	
4 bits de PID	4 bits de PID en complemento
8 bits	

Tabla 13. Estructura de un paquete de tipo *handshake*

2.3.4.5 Paquete especial PRE

Este tipo de paquetes es un caso especial para la transmisión en modo de baja velocidad. Como se analizará en el siguiente apartado, los dispositivos de baja velocidad sólo pueden hacer transferencias tipo control e interrupción, por lo que para identificar este tipo de transacciones de baja velocidad se implementa el uso de este paquete especial *PRE*. Una característica importante es que el paquete *PRE* es de tipo prefijo por lo que sustituirá al *Token SOF*, por lo que un paquete de un dispositivo de baja velocidad nunca tendrá un paquete *SOF*. En la Tabla 14 se muestra la estructura de este tipo de paquete.

PRE	
4 bits de PID	4 bits de PID en complemento
8 bits	

Tabla 14. Estructura de un paquete de tipo *Especial*

2.3.5 Tipos de transacción

Una secuencia definida de paquetes es usada para definir el movimiento de los datos entre el *Host* y el dispositivo USB de Entrada/Salida. Cada transacción **debe** ocurrir dentro de un mismo *frame*. No se permite el envío de una transacción que abarque múltiples *frames* (esto cambia para velocidades *HIgh-*

Speed en la versión 2.0 del estándar USB, pero no será analizado en este trabajo). Una vez que una secuencia se inició, debe ser completada sin la intervención de los paquetes de otras transacciones. La especificación define cuatro tipos de transacción para los distintos tipos de datos que pueden ser transmitidos por el bus. Todas las transacciones usan el mismo bloque de construcción para los paquetes, sólo se diferencian en el modo en que son programados y la manera en que responden a un error. El software del *Host* en la PC debe lidiar con dos parámetros: tiempo de entrega y calidad de entrega. Estos parámetros son dependientes del tipo de datos a transferir. La Tabla 15 muestra los modos de transferencia existentes tanto para la versión 1.0 como para la 2.0 del estándar, así como las características principales de cada modo.

Tipo de Transacción	Parámetro atendido	Tamaño Máximo del Paquete (Bytes)		
		Low Speed	*Full Speed*	*High Speed*
Interrupción	Calidad	<8	<64	<3072
Bulk	Calidad	N/A	8,16,32,64	<512
Isócrona	Tiempo	N/A	<1023	<3072
Control	Tiempo y Calidad	8	8,16,32,64	64

Tabla 15. Características de los modos de transferencia de datos

El *Host* en la PC, garantiza tiempos de entrega ya que reserva *frames* para transferencias de control e isócronas. El *Host* también garantiza calidad en entrega mediante el mecanismo de *handshake*.
En general:

- Si los datos son recibidos correctamente, se genera un ACK,
- Si hay algún problema con la transmisión se envía un NAK,
- Si el receptor está confundido, se transmite un STALL.

A continuación se describen sólo los modos *Bulk* y control que serán los modos utilizados en este trabajo. Con fondo blanco se ejemplifican los paquetes generados por el *Host* y los paquetes transmitidos por el **dispositivo** se rellenan con **negro**, en las tablas mostradas a continuación. Así mismo, HS indica que se trata de un paquete de tipo *handshake*.

2.3.5.1 Transferencias en modo bulk

La transferencia en este modo, garantiza calidad en entrega, es decir, **sin** pérdida de datos, es enviada una vez que todas las transferencias programadas y con garantía de tiempo hayan sido entregadas. Es decir, si se hace una transferencia *Bulk* en un bus USB muy ocupado, esta tardará más tiempo; por otro lado si se hace una transferencia *Bulk* en un canal con tiempo disponible, el *Host* puede programar múltiples transferencias en un *frame*. Y los paquetes DATA0 y DATA1 se irán alternando para verificación de errores[12].

[12] HYDE, John. *"USB Design by Example"* p. 30

Si un dispositivo no pudiera recibir la transmisión, enviará un NAK. En este caso el *Host* reintentará la transmisión del paquete en el siguiente *frame*.

Secuencia1	**IN**	Dirección	Endpoint	CRC5	DATA1	Datos	CRC16	ACK	
	Paquete *Token*				Paquete datos			HS	
Secuencia2	**IN**	Dirección	Endpoint	CRC5	NAK				
	Paquete *Token*				Paquete *Handshake*				
Secuencia3	**OUT**	Dirección	Endpoint	CRC5	DATA1	Datos	CRC16	ACK	
	Paquete *Token*				Paquete datos			HS	

Tabla 16. Ejemplificación de la transmisión de paquetes

En la secuencia 1, mostrada en la Tabla 16, se ilustra una transferencia exitosa del dispositivo hacia el *Host*. El ACK mostrado es una respuesta positiva en la entrega válida de los datos. Si la respuesta fuera muy rápida, es posible que el dispositivo aún no tenga los datos listos para enviar al *Host*, por lo que enviará un NAK, como se muestra en la secuencia 2, y el *Host* reintentaría la lectura en el siguiente *frame*. La tercera secuencia muestra una transferencia exitosa hacia el dispositivo desde el *Host*. Si hubiera alguna confusión en los comandos recibidos por el paquete *Token*, el dispositivo generaría un paquete *handshake* STALL.

2.3.5.2 Transferencias de control

Este tipo de transferencias son las más complicadas, ya que necesitan mucha protocolización para asegurar que los comandos son entregados tanto en tiempo como en calidad. Este tipo de transferencias son utilizadas generalmente sólo en el momento de enumerar un dispositivo (la enumeración será explicada más adelante), por lo que no representa un problema para el desempeño del sistema. Una transferencia de control está dividida en tres etapas, las cuales utilizan las construcciones de paquetes antes analizadas. Todas las transmisiones de control comienzan con la fase de configuración y terminan con la etapa de estado, y una etapa de datos que es opcional. Todas las transferencias de control apuntan a la dirección del *Endpoint 0* en el dispositivo que se tiene como objetivo[13].

La fase de configuración consiste en el envío de tres paquetes, SETUP, DATA0 y uno de *handshake*, como se muestra en la Tabla 17.

[13] HYDE, John. *"USB Design by Example"* p. 32

SETUP	Dirección	Endpoint	CRC5	DATA0	8 Bytes	CRC16	ACK
Paquete *Token*				Paquete datos			H.S.

Tabla 17. Fase de configuración de transferencia de control

El paquete DATA0 en esta etapa siempre contiene 8 bytes, y los datos contenidos en el paquete están predefinidos (se analizarán en la siguiente sección). El paquete de *handshake* es siempre ACK, debido a que un paquete SETUP siempre debe ser recibido, incluso si implica abortar otra transmisión de control. En esta etapa también se especificará si es requerida una etapa de datos, y si serán de lectura o escritura, haciendo uso de paquetes IN y OUT.

Cualquier transferencia de control siempre termina con la fase de status o estado si:

- No hay etapa de datos, el dispositivo sólo envía ACK
- La etapa de datos es de lectura, el Host deberá enviar ACK
- La etapa de datos es de escritura, el dispositivo deberá enviar ACK.

La etapa de estado es un paquete IN si el dispositivo provee el *status*, y será un OUT si el *Host* es el que provee el estado. Un paquete de datos de tamaño cero es usado para indicar éxito, y se contestará ACK, NAK o STALL según sea la respuesta.

La Tabla 18 muestra un ACK enviado por el dispositivo en la fase de estado:

IN	Dirección	Endpoint	CRC5	DATA1	0 bytes	CRC16	ACK
Paquete Token				Paquete Datos			H.S.

Tabla 18. Transferencia de control IN exitosa

La Tabla 19 muestra un ACK desde el Host en la etapa de status:

OUT	Dirección	Endpoint	CRC5	DATA1	0 bytes	CRC16	ACK
Paquete Token				Paquete Datos			H.S.

Tabla 19. Transferencia de control OUT exitosa

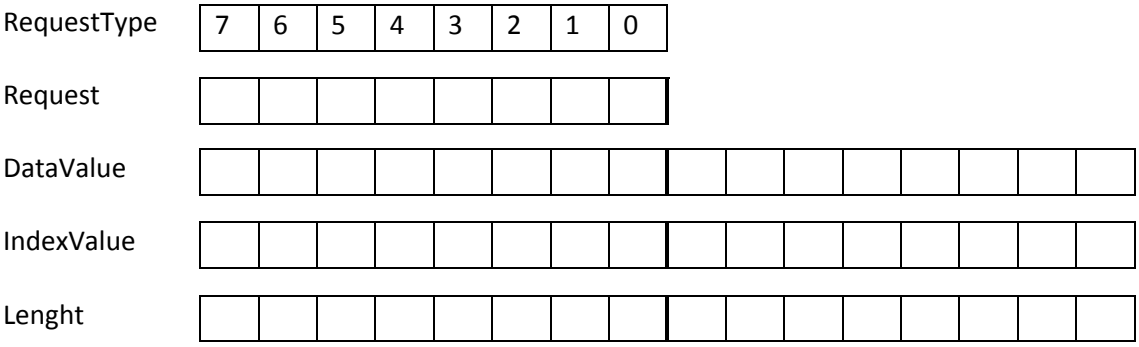
Por último, en la Tabla 20 se muestra un NAK desde el dispositivo en la parte de estado:

IN	Dirección	Endpoint	CRC5	NAK
Paquete Token				H.S.

Tabla 20. Transferencia de control IN fallida

Peticiones en transferencias de control

Las peticiones que se envían en el paquete de datos DATA0 en la fase de configuración, se forman de la siguiente manera (hay que recordar que el paquete de datos es de 8 bytes).



Definición del campo de bits de RequestType(Tipo de petición)

Bit 7:	0=Transferencia hacia dispositivo		1=Transferencia desde Dispositivo	
Bit 6,5:	00=estándar	01=clase	10=fabricante	11=reservado
Bit 4-0:	00000=dispositivo	00001=interfaz	00010=*endpoint*	00011=otro
	El resto de los códigos está reservado.			

El bit 7 del campo *RequestType* indica la dirección de la siguiente transacción.

Los bits 6 y 5 indican cómo debe ser interpretado el byte *Request.*

Los bits bajos indican el destino de esta petición (dispositivo, interfaz o *endpoint*).

El campo *Request*, le indica al dispositivo la información que el *Host* requiere. A continuación se muestran las peticiones de tipo estándar a un **dispositivo** USB.

Valor	Petición	Acción Requerida
0	Get_Status	Devuelve el estado actual
1	Clear_Feature	Reinicializa la característica indicada
3	Set_Feature	Establece la característica indicada
5	Set_Address	Almacena la dirección USB y se usará desde ese instante
6	Get_Descriptor	Devuelve el descriptor pedido
7	Set_Descriptor	Establece el descriptor indicado
8	Get_Configuration	Devuelve configuración actual o 0 sino ha sido configurado
9	Set_Configuration	Establece la configuración a la indicada
10	Get_Interface	Devuelve la interfaz actual
11	Set_Interface	Establece la interfaz a la especificada
12	Sync_Frame	Sincronizar números de los *frames* USB (Dispositivos asíncronos)

Tabla 21. Tipos de peticiones estándar hacia un dispositivo USB

Los 6 bytes restantes son utilizados por las peticiones que requieren de datos extra. Si se requiere un byte o una palabra en el campo *DataValue*, se hace con *offset* 2 y/o 3. Si se requiere un valor de índice

se hace con el *offset* 4 y/o 5 para el campo *IndexValue*. En el campo *Length* con offset 6 y 7, se debe especificar el tamaño de la transferencia de datos, que viene a continuación en el paquete de datos.

2.3.6 Enumeración

Se le llama enumeración al proceso que identifica y asigna una dirección única a los dispositivos conectados al bus[14]. Debido a que el estándar permite la conexión y desconexión de dispositivos, el proceso de enumeración es una actividad continua y es desarrollada por el software en el *Host*.

Cuando un dispositivo se conecta a un puerto disponible, las siguientes acciones son ejecutadas, como nota todas son iniciadas por el *Host*.

1. El *hub* al que es conectado, le es requerido el estado del puerto, de esta manera el *Host* descubre el nuevo dispositivo conectado al bus. En este momento el dispositivo ya está alimentado, pero el puerto está deshabilitado.
2. Ya que se conoce el puerto al que se conectó, el *Host* espera 100 ms a que se termine de conectar el dispositivo al puerto y se estabilice la alimentación, después el *Host* envía una habilitación al puerto y un *reset* al puerto.
3. El *hub* en el que está el puerto realiza la reinicialización requerida para él. Cuando la señal de *reset* se termina, el puerto se encuentra debidamente habilitado, y el dispositivo se encuentra en el modo predeterminado, es decir, no puede consumir más de 100 mA del bus.
4. El *Host* asigna una dirección única al dispositivo, modificando el estado del dispositivo a dirección.
5. El *Host* lee el descriptor del dispositivo para determinar cuál es la máxima velocidad de transferencia que puede utilizar.
6. El *Host* lee todas las configuraciones que el dispositivo pueda tener. Este proceso puede tardar varios milisegundos dependiendo de la cantidad de configuraciones que tenga. Por el lado del software el *Host* carga en este momento los controladores necesarios para el dispositivo.
7. Basado en la información de configuración y cómo el dispositivo debe ser usado, el *Host* asigna un valor de configuración al dispositivo. En este momento el dispositivo se encuentra en estado configurado y todos los *endpoints* definidos en su configuración, se encuentran accesibles, y también puede obtener la corriente configurada en su descriptor. En otras palabras, el dispositivo está listo para usarse.

Cuando el dispositivo es removido, el *hub* al que se encuentra conectado envía una señal al *Host*, que inmediatamente deshabilita el puerto y actualiza su información topológica.

[14] USB-IF. *"Universal Serial Bus Revision 2.0 specification"* p. 20

2.3.7 Descriptores

Un dispositivo USB reporta sus capacidades usando descriptores[15]. Un descriptor es una estructura de datos con un formato bien definido. Cada descriptor inicia con campos de distintos tamaños que indican el número total de bytes contenidos en el descriptor y el tipo de descriptor.

El uso de descriptores asegura que se almacenarán de manera correcta los atributos individuales de las configuraciones, ya que cada configuración puede tener su propio descriptor o reusar descripciones de otros descriptores de otras configuraciones.

Un dispositivo USB puede devolver los descriptores de clase específica o de constructor específico de dos maneras:

- Si el descriptor específico usa el mismo formato que un descriptor estándar, el descriptor específico de clase o constructor deberá ser transmitido inmediatamente después del descriptor estándar que vaya a modificar.
- Si el descriptor específico es independiente o no usa un formato estándar. Se deberá definir la manera correcta para poder obtener estos descriptores.

Este tipo de descriptores no estándar no serán usados en este trabajo por lo que no serán analizados más a fondo.

2.3.7.1 Dispositivo

Este tipo de descriptor informa las características generales acerca del dispositivo, es decir, información que implica globalmente el funcionamiento del dispositivo. Un dispositivo USB *sólo* tiene *un* descriptor de dispositivo. La Tabla 22 muestra la composición de un descriptor de dispositivo.

Offset	Campo	Bytes	Valor	Descripción
0	bLength	1	Número	Tamaño del descriptor en bytes (18)
1	bDescriptorType	1	Constante	Descriptor dispositivo (0x01)
2	bcdUSB	2	BCD	Versión codificada en BCD (p.e. 0x0200, para la versión 2.00)
4	bDeviceClass	1	Clase	Código de clase, predefinidos por el USB-IF para los valores entre 0x01 y 0xFE. Si es ***cero***, entonces cada interfaz definirá su clase dentro de una configuración. Si este valor es 0xFF entonces se usará una clase específica por el fabricante
5	bDeviceSubClass	1	Subclase	Código de subclase, predefinidos por la USB-IF para cualquier valor distinto de 0xFF, que representa el específico

[15] USB-IF. *"Universal Serial Bus Revision 2.0 specification"* p. 260

				por el fabricante
6	bDeviceProtocol	1	Protocolo	Código de protocolo, predefinido por la USB-IF. Si está en 0xFF implica un protocolo definido por el fabricante
7	bMaxPacketSize0	1	Número	Tamaño máximo del paquete para el EP0 (sólo 8, 16, 32 ó 64 son válidos)
8	idVendor	2	ID	Identificador de fabricante (asignado por la USB-IF)
10	idProduct	2	ID	Identificador de producto (asignado por el fabricante)
12	bcdDevice	2	BCD	Versión del dispositivo en BCD
14	iManufacturer	1	Índice	Índice del descriptor de cadena para el fabricante
15	iProduct	1	Índice	Índice del descriptor de cadena para encontrar el producto
16	iSerialNumber	1	Índice	Índice en el descriptor de cadena para el número de serie
17	bNumConfig	1	Número	Número de configuraciones posibles

Tabla 22. Estructura del descriptor de dispositivo

Las clases fueron diseñadas en el estándar para facilitar la escritura de software para los dispositivos en el *Host*, en algunos casos no es necesario el desarrollo de ningún software, debido a que ya vienen incluidos en el sistema operativo. Algunos ejemplos de clases incluyen: “Interfaz Humana”, “Almacenamiento”, “Sonido”, “Alimentación”, etc. El protocolo es usado en algunas clases para su comunicación. Ahora bien, si un dispositivo USB es parte de alguna clase predefinida, el sistema operativo tendrá listo un controlador para la interfaz, y no será necesario desarrollar ninguno para este dispositivo. Otro aspecto importante del estándar es que es independiente de la plataforma, es decir, puede funcionar transparentemente en sistemas Linux, Mac o Windows, por mencionar algunos.

2.3.7.2 Configuraciones

Un dispositivo USB puede tener una gran cantidad de configuraciones, pero generalmente un dispositivo tiene sólo una, ya que generalmente son simples, así como una configuración puede tener varias interfaces a su vez, una interfaz puede tener otro número de *endpoints*. El descriptor de configuración contiene información, por ejemplo, si requiere alimentación externa o si se alimenta del bus, la cantidad de corriente que consume, el número de interfaces que tiene, etc. Una vez que el dispositivo ha sido configurado, ya no es posible hacer modificaciones sustanciales a la configuración. Si una interfaz tiene configuraciones alternativas, puede hacerse un cambio una vez que el dispositivo ya ha sido configurado. La Tabla 23 muestra todos los campos que contiene la estructura para el descriptor estándar de configuración.

Offset	Campo	Bytes	Valor	Descripción
0	bLength	1	Número	Tamaño del descriptor en bytes (9)
1	bDescriptorType	1	Constante	Descriptor configuración (0x02)
2	wTotalLength	2	Número	Tamaño total devuelto para esta configuración, incluyendo el tamaño combinado de todos los descriptores de esta configuración (configuración, interfaz, *endpoint* y específicos de clase o configuración)
4	bNumInterfaces	1	Número	Número de interfaces suportadas por esta configuración
5	bConfigurationValue	1	Número	Valor para usar como argumento cuando el *Host* llame a *SetConfiguration*() en la enumeración, para seleccionar esta configuración
6	iConfiguration	1	Índice	Índice en el descriptor de cadena para la descripción de esta configuración
7	bmAttributes	1	Mapa de bits	Características de Configuración D7: Reservado (siempre 1) D6: Alimentado por bus D5: *Remote Wakeup*(Encendido Remoto) D4…0: Reservado (ceros) Si D6 está en 1, entonces deberá proporcionarse la cantidad de corriente necesaria en bMaxPower
8	bMaxPower	1	mA	Máximo consumo de corriente en el bus USB, cuando el dispositivo está funcionando normalmente, expresado en unidades de 2 mA. Por ejemplo 500 mA = 250

Tabla 23. Estructura del descriptor de configuración

2.3.7.3 Interfaz

Este descriptor describe una interfaz específica dentro de una configuración. Una configuración provee una o más interfaces, cada una con ninguno o más *endpoints* describiendo un conjunto único de *endpoints* dentro de una configuración. La Tabla 24 contiene el resumen de los elementos del descriptor de Interfaz.

Offset	Campo	Bytes	Valor	Descripción
0	bLength	1	Número	Tamaño del descriptor en bytes (9)
1	bDescriptorType	1	Constante	Descriptor de interfaz (0x04)
2	bInterfaceNumber	1	Número	Índice de la interfaz expuesta
3	bAlternateSetting	1	Número	Número utilizado para seleccionar una interfaz alternativa usando su índice
4	bNumEndpoint	1	Número	Cantidad de *endpoints* contenidos en la interfaz
5	bInterfaceClass	1	Clase	Código de clase. 0xFF para usar una clase

				definida por el constructor. 0x00 para uso futuro, cualquier otro valor es asignado por la USB-IF para el tipo de dispositivo
6	bInterfaceSubClass	1	Subclase	Código de subclase, asignado de igual manera que InterfaceClass
7	bInterfaceProtocol	1	Protocolo	Código de protocolo asignado por el USB-IF según bInterfaceClass y bInterfaceSubClass. Si es un protocolo definido por el constructor se deberá asignar 0xFF
8	bInterface	1	Índice	Índice a utilizar en el descriptor de cadena para encontrar la descripción de esta interfaz

Tabla 24. Estructura del descriptor de interfaz

2.3.7.4 Endpoint

Cada *endpoint* tiene su descriptor en una interfaz. Este descriptor contiene la información requerida por el *Host* para determinar el ancho de banda requerido para cada *endpoint*. Nunca hay un descriptor para el *endpoint* cero. En la Tabla 25 se muestra el descriptor de un e*ndpoint*.

Offset	Campo	Bytes	Valor	Descripción
0	bLength	1	Número	Tamaño del descriptor en bytes (7)
1	bDescriptorType	1	Constante	Descriptor de Endpoint (0x05)
2	bEndpointAddress	1	Endpoint	La dirección del *endpoint* Bits 3..0 Número de Endpoint Bits 4..6 Reservados (Ceros) Bit 7 Dirección, ignorado por *endpoints* de control 0: Endpoint OUT 1: Endpoint IN
3	bmAttributes	1	Mapa de bits	Bits 1..0: Tipo de transferencia 00 Control 01 Isócrona 10 *Bulk* 11 Interrupción Si la transferencia no es isócrona, los bits de 7 al 2, deberán ir establecidos a cero. De lo contrario deberán definirse de cierta manera, debido a que no se usará este tipo de *endpoint*, el mapa de bits no será descrito
4	wMaxPacketSize	2	Número	Tamaño máximo de paquete que este *endpoint* es capaz de enviar o recibir
6	bInterval	1	Número	Intervalo para obtener las transferencias de datos, usado sólo para transferencias de tipo interrupción. Es un valor que debe estar entre 1 y 255, este valor está definido en conteo de *frames*

Tabla 25. Estructura de un descriptor de *endpoint*

2.3.7.5 Cadena

Este tipo de descriptores contiene información entendible para el usuario, y es opcional. Si un dispositivo no soporta este tipo de descriptor todas las referencias al descriptor de cadena deberán ser cero. Este descriptor usa codificación *unicode*, para que puedan soportar múltiples lenguajes. La estructura del descriptor cero de cadena es la mostrada en la Tabla 26, donde deberán incluirse los idiomas que se desea soportar en el dispositivo.

Offset	Campo	Bytes	Valor	Descripción
0	bLength	1	N+2	Tamaño del descriptor en bytes
1	bDescriptorType	1	Constante	Descriptor *string* (0x03)
2	wLANGID[0]	2	Número	Código Lenguaje 0, soportado
…	…	…	…	…
N	wLANGID[N-2]	2	Número	Código Lenguaje N-2, soportado

Tabla 26. Estructura del descriptor de cadena principal

Deberán existir tantos consecutivos descriptores como lenguajes definidos en el descriptor de cadena principal o cero y tendrán la estructura mostrada en la Tabla 27.

Offset	Campo	Bytes	Valor	Descripción
0	bLength	1	Número	Tamaño del descriptor en bytes
1	bDescriptorType	1	Constante	Descriptor *string* (0x03)
2	bString	x	Unicode	Cadena codificada en *Unicode*

Tabla 27. Estructura del descriptor de cadena para cada lenguaje

2.4 Control remoto

Este tipo de control permite manipular a distancia actuadores, motores, y hasta la misma computadora que se encuentra conectada a una red. Para poder llevar a cabo esta tarea es necesario apoyarse en herramientas que ya han sido generadas y aprovecharlas, un excelente método ya desarrollado y probado es utilizando las llamadas aplicaciones remotas a través de servidores de Internet, usadas en aplicaciones tales como comercio electrónico, tiendas virtuales, enseñanza interactiva, aplicación de encuestas en línea, etcétera. Este tipo de programas se ejecutan enteramente en el servidor anfitrión, es decir, no dependen del cliente ni requieren ningún requisito especial, más que una conexión a Internet y un navegador. Esto trae una ventaja muy grande, ya que cualquier cliente que se conecte al servidor no requerirá tener instalado un sistema operativo específico, una aplicación especial, ni tampoco un explorador de Internet en especial, por lo que el acceso se hace muy flexible.

2.4.1 Arquitectura

La arquitectura usada en el laboratorio con los sistemas de control y las interfaces es la mostrada en la Figura 9 de manera conceptual.

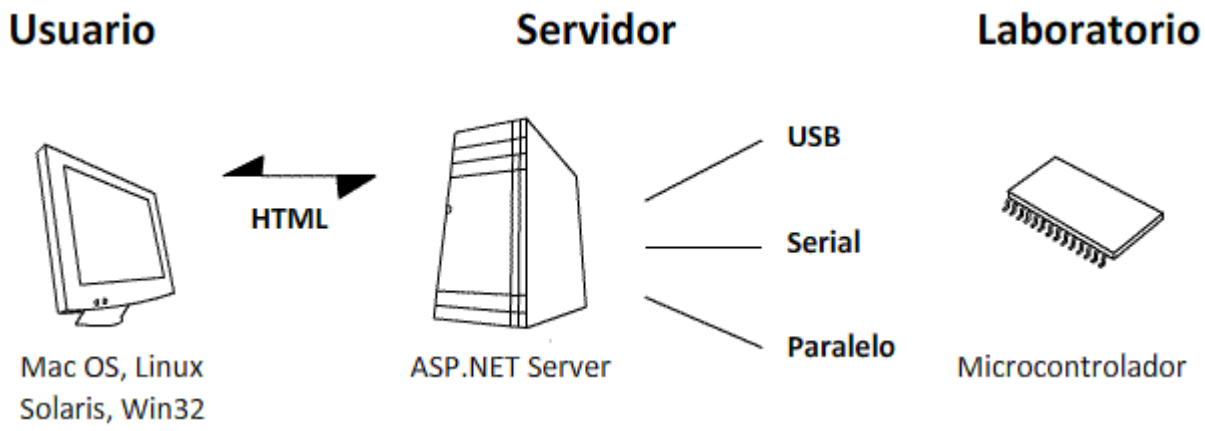


Figura 9. Arquitectura del laboratorio y el control remoto

Como puede observarse, el usuario sólo requiere un equipo con conexión a Internet; el servidor se encargará se comunicarse con el usuario, haciendo uso del lenguaje HTML, que es un estándar. En páginas de Internet, el servidor generará todo el contenido de manera que el usuario pueda verlo según sus necesidades y los datos que haya enviado, y de igual manera se encargará de comunicarse con las interfaces u otros dispositivos que se colocarán en el laboratorio, mediante los puertos paralelo, serial y USB.

2.4.2 Aplicaciones distribuidas

El modo de trabajar de este esquema es simple, y se le conoce como **Aplicación Distribuida**, es decir, la ejecución de la aplicación o el programa se propaga a través de más de un equipo, para así mejorar el desempeño de la aplicación[16], donde el cliente, el usuario, no requiere tener una aplicación instalada en el equipo, lo que conlleva muchas ventajas, por ejemplo:

- Se evita tener una cantidad incontrolable de versiones de una misma aplicación
- Todos los clientes tienen la versión más reciente, es decir la que contiene menos errores
- Al ser una aplicación web, no requiere de múltiples compilaciones para distintas plataformas
- Se distribuye el trabajo en distintas capas
- Si falla una capa, el resto puede seguir funcionando, evitando así una caída total de la aplicación.

Generalmente las aplicaciones que se encargan de los servicios y aplicaciones de Internet, se encargan de toda la parte de comunicación a través de la red, y la detección y corrección de errores, ya que se basan en protocolos de red robustos; además proveen un método de programación sencillo, por lo que son una elección importante para el desarrollo acelerado de aplicaciones sobre Internet.

[16] MC DONALD, Matthew. *"Microsoft® .NET Distributed Applications: Integrating XML Web Services and .NET Remoting".*

Capítulo 3

Diseño e implementación

En este capítulo se presentan las posibles soluciones del circuito requerido para la comunicación mediante el puerto USB, así como el desarrollo del controlador de hardware para Windows y además el diseño de los dispositivos de entrada/salida para los puertos serial y paralelo; también se incluye la implementación de la aplicación sobre Internet para el control remoto.

3.1 Puerto USB

El puerto USB fue el desarrollo que tomó más tiempo, tanto en diseño como en implementación, debido a que este puerto, a diferencia del serial y el paralelo, no puede ser accedido directamente. Debe de contarse con un sistema operativo que tenga soporte para USB, que se encarga de todas las peticiones de entrada y salida hacia y desde los puertos USB, y posteriormente debe de lidiarse con las implementaciones dependientes del sistema operativo. En el caso de este trabajo, se utiliza el sistema operativo Windows de Microsoft, ya que es ampliamente usado y porque se cuenta con todas las facilidades para trabajar con éste.

3.1.1 Requerimientos

Primeramente fue necesario definir exactamente lo que se requería, y evitar el uso de los puertos serial y paralelo para nuevos diseños, ya que estos puertos se consideran obsoletos, por lo que la primera opción fue utilizar el puerto USB, ya que fue diseñado para conectar una gran variedad de dispositivos y ofrece una gran velocidad de transferencia de datos. Una segunda opción fue el puerto *Firewire*, pero a diferencia del puerto USB, aquél fue diseñado para aplicaciones de audio y video, es más, difícilmente puede ser encontrado en los equipos de cómputo, y los protocolos de comunicación son de mayor complejidad, por lo que los microcontroladores capaces de manejar estos protocolos son de mayor costo, lo que provocó que se descartara su uso.

Por otro lado, una ventaja extra de utilizar el puerto USB es que ha crecido en popularidad. En las computadoras actuales ha sustituido progresivamente a los puertos serial y paralelo, y es imposible ver un equipo nuevo que no contenga al menos 2 ó 4 puertos USB de alta velocidad.

Una vez seleccionado el puerto USB, fue necesario decidir qué modo de transmisión era necesario utilizar; un aspecto a tomar en cuenta era que no tenía caso hacer un esfuerzo en desarrollar un proyecto como éste, si no iba a generar suficientes ventajas, el uso de alta velocidad del puerto fue un factor fundamental, por lo que se seleccionó un balance entre velocidad, robustez y costo. Para lograr este balance se decidió el uso del puerto USB con dos *Endpoints* tipo *Bulk*, uno de entrada y otro de salida, en modo *Full-Speed*, como se analizó en el capítulo anterior, el modo *Full-Speed* ofrece velocidades de hasta 12 Mbps y los circuitos que facilitan el uso de este modo son de bajo costo, los *endpoints* tipo *bulk* aseguran la entrega, es decir robustez en la transmisión.

3.1.2 Transceptores USB disponibles

En el mercado hay una gran variedad de transceptores[17] USB, con distintas capacidades, características y precios, se analizaran algunos de los más aptos para el desarrollo de este trabajo.

Familia Cypress EzUSB

Los dispositivos de la familia EZ-USB FX (CY7C646xx), son dispositivos USB *Full-Speed*, que pueden tener hasta 32 *endpoints* de cualquier tipo, lo cual permite una amplia gama de desarrollos; tienen integrado un transceptor I^2C que facilita la intercomunicación entre microcontroladores a altas velocidades, lo cual permitiría desarrollar el proyecto en más de un microcontrolador. Tienen integrados puertos seriales, y tienen memoria de 8 KB, lo cual es suficiente para un programa como el que se desarrollará. Como se mencionó, esta familia tiene integrado un microcontrolador de propósito general, lo que es una ventaja extra ya que no se tendrían dos dispositivos, un microcontrolador para el control de motores, actuadores, comunicaciones seriales, y un transceptor USB que se encargue de la comunicación con la PC usando el puerto USB.

Familia National Instruments USBN96xx

Son transceptores USB *Full-speed*, soportan 3 endpoints de entrada y 3 de salida, suficientes para el desarrollo de este trabajo, tienen soporte *DMA*[18] para control con microprocesador externo; la desventaja de esta solución es que se requiere de un microcontrolador que haga el resto de las funciones, ya que este circuito sólo se encarga de la comunicación USB.

Dispositivos Philips PDIUSB1x

Estos dispositivos también son transceptores USB *Full-Speed* que soportan hasta 6 *endpoints y* de igual manera, se satisfacen los requerimientos USB del proyecto; cuentan con la misma desventaja que los de National Instruments, al requerir un microcontrolador externo, pero a

[17] Un *transceptor* es aquel dispositivo que es capaz tanto de transmitir como de recibir datos utilizando ciertos métodos o protocolos.

[18] DMA (*Direct Memory Access*), Acceso directo a memoria, es una característica en los sistemas que permite el acceso a la memoria principal sin requerir de tiempo ni permiso de la unidad central de proceso.

diferencia de estos, cuentan con una interfaz I^2C que es comúnmente encontrada en la mayoría de los microcontroladores y microprocesadores.

Familia Microchip PIC18Fx550

Estos dispositivos son microcontroladores con transceptor USB integrado, ofrecen una amplia gama de funcionalidades, ya que además integran convertidores analógicos Digitales de 10 bits, contadores, interfaz I^2C, controladores PWM, comparadores, puertos seriales, y además tienen memoria f*lash* para el código, así como memoria RAM y EEPROM. Con todo lo anterior es uno de los dispositivos de costo bajo, vienen en formato DIP, por lo que es fácil de utilizar.

Tras lo anterior, la selección final fue el uso de dispositivos Microchip PIC18F4550, que ofrecen hasta 32 KB de memoria *flash* para programa, que son suficientes para el desarrollo actual y futuras modificaciones. Lo más importante de este circuito es que integra muchas características útiles para el desarrollo de cualquier aplicación en el laboratorio, y una ventaja extra es que se tiene experiencia previa con dispositivos Microchip, por lo que se cuenta con los programadores, compiladores y depuradores necesarios para el desarrollo del proyecto con este microcontrolador.

En el mercado hay más dispositivos con transceptor USB integrado, pero son para aplicaciones particulares, es decir, son soluciones basadas en los circuitos presentados y vienen preprogramados para solucionar un problema específico, por lo que no son una opción para el proyecto.

3.1.3 Circuito USB

Ahora bien, ya que se sabe con certeza que se usará un microcontrolador Microchip USB *Full-Speed* y que se requieren dos endpoints tipo *bulk*, un generador y otro consumidor de datos. Se procederá al diseño tanto físico como de software para el proyecto, no sin antes mostrar las generalidades del microcontrolador seleccionado.

3.1.3.1 Microcontrolador Microchip PIC18F4550

Este circuito tiene las siguientes funcionalidades USB:

- Compatible con USB 2.0
- Transceptor USB integrado con regulador de voltaje
- Soporta *Low Speed* (1.5 Mbps) y *Full Speed*(12 Mbps)
- Soporta transferencias *Bulk*, Isócronas, Interrupción
- Soporta hasta 32 endpoints

Y además tiene los siguientes periféricos integrados:

- 2 Módulos PWM/ Comparadores de 16 bits
- Puertos Seriales

- Módulo I^2C Maestro y esclavo
- 13 canales para convertidor analógico digital de 10 bits de resolución
- 5 puertos de entrada/salida

3.1.3.2 Diseño electrónico

El diseño se basó en el sugerido por el fabricante, con las modificaciones necesarias para adaptarlo al proyecto. El diagrama general para conexión del microcontrolador se muestra en la Figura 10.

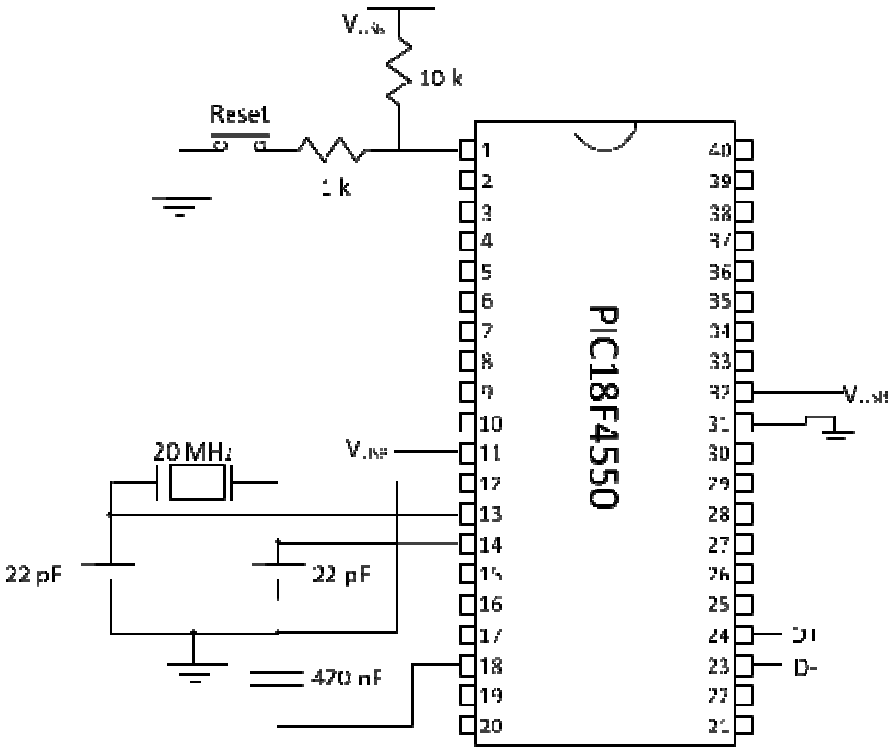


Figura 10

Este es el circuito básico para poder iniciar el trabajo con el dispositivo USB, al que pueden agregarse elementos para utilizar otras funcionalidades contenidas en el microcontrolador, tales como los convertidores Analógico Digitales, los módulos PWM para controlar motores, o cualquier periférico analizado en el apartado anterior.

Este circuito puede ser conectado al puerto USB de cualquier PC, ahora bien, en caso de que el microcontrolador no contuviera ninguna información y se conectara a una PC, se generaría un mensaje del sistema operativo, tal como "Dispositivo USB no reconocido", por lo que es necesario crear el firmware o el software que irá almacenado en la memoria de programa del microcontrolador, que contendrá todas las capacidades del dispositivo que se está creando para que sean anunciadas al sistema *Host* cuando éste sea conectado, así como el programa que permita crear cualquier funcionalidad que utilice los periféricos del microcontrolador, una vez que éste sea enumerado.

3.1.4 Desarrollo firmware

Como se comentó, el firmware es el programa que se encuentra almacenado en el dispositivo, es decir, es el programa insertado en cualquier aparato electrónico por el fabricante, para proveer funcionalidades en un producto específico.

El desarrollo del firmware está desarrollado en lenguaje C para el PIC18F4550; lo primero que se debe hacer es anunciar al sistema *Host* que el dispositivo tiene dos *endpoints* de tipo *bulk*, que es la parte más importante, pero antes de esto es necesario proveer los descriptores, que fueron analizados anteriormente. A continuación se analizará cómo se implementa lo anterior[19].

En lenguaje C, los descriptores se almacenan en simples arreglos de cadenas, donde se contiene la información analizada en el capítulo anterior.

Descriptor de dispositivo

```
//Descriptor de Dispositivo
char const USB_DEVICE_DESC[] ={
    USB_DESC_DEVICE_LEN,  //Tamaño del descriptor
    0x01,                 //Constante para descriptor de dispositivo (0x01)
    0x10, 0x01,           //Versión USB en BCD
    0x00,                 //Código de Clase
    0x00,                 //Código de Sub Clase
    0x00,                 //Código de Protocolo
    USB_MAX_EP0_PACKET_LENGTH,   //Tamaño del Paquete para el Endpoint0
    0x3f, 0x44,           //Identificador de Fabricante (Puede ser cualquier)
    0xff, 0x00,           //Identificador de Producto (Puede ser cualquier)
    0x01, 0x00,           //Versión del dispositivo
    0x01,                 //Índice en el descriptor de cadena para Fabricante
    0x02,                 //Índice en el descriptor de cadena para Producto
    0x00,                 //Índice en el descriptor de cadena para el # de serie
    USB_NUM_CONFIGURATIONS//Número de configuraciones del dispositivo (1)
 };
```

[19] Si se desconocen los términos utilizados y porqué el orden utilizado, revisar la sección 2.3.7 de este trabajo.

Descriptor de configuración, interfaces, y endpoints

```
//descriptor de configuración
char const USB_CONFIG_DESC[] = {
   //descriptor de configuración para índice 1
   USB_DESC_CONFIG_LEN,     //Tamaño del descriptor (9)
   USB_DESC_CONFIG_TYPE,    //Constante para el descriptor de configuración (0x02)
   USB_TOTAL_CONFIG_LEN, 0,      //Tamaño total de los descriptores
   1,                       //Número de interfaces soportadas
   0x01,                    //Identificador para esta configuración
   0x00,                    //Índice en el descriptor de cadena para esta configuración
   0xC0,                    //Alimentado por bus
   0x32,                    //Corriente requerida por el dispositivo (0x32=50d=100[mA])

//descriptor de interfaz 0 configuración alternativa 0
   USB_DESC_INTERFACE_LEN,  //Tamaño del descriptor (9)
   USB_DESC_INTERFACE_TYPE, //Constante para el descriptor de interfaz (0x04)
   0x00,                    //Identificador para esta interfaz
   0x00,                    //Configuración alternativa
   0x02,                    //Cantidad de endpoints en esta interfaz (sin contar EP0).
   0xFF,                    //Código de clase, 0xFF = definido por fabricante
   0xFF,                    //Código de subclase, 0xFF = definido por fabricante
   0xFF,                    //Código protocolo, 0xFF = definido por fabricante
   0x00,                    //Índice en el descriptor de cadena para esta interfaz

//descriptor de endpoint
   USB_DESC_ENDPOINT_LEN, //Tamaño del descriptor (7)
   USB_DESC_ENDPOINT_TYPE,//Constante para el descriptor de endpoint (0x05)
   0x81,                    //Número de endpoint y dirección (0x81 = EP1 IN)
   0x02,                    //Tipo de transferencia (2 es bulk)
   USB_EP1_TX_SIZE & 0xFF, USB_EP1_TX_SIZE >> 8,  //Tamaño máximo de paquete
   0x01,                    //intervalo de lecturas en [ms] (Para transferencias
Isócronas)

//descriptor de endpoint
   USB_DESC_ENDPOINT_LEN, //Tamaño del descriptor
   USB_DESC_ENDPOINT_TYPE,//Constante para descriptor de endpoint (0x05)
   0x01,                    //Número de endpoint y dirección (0x01 = EP1 OUT)
   0x02,                    //Tipo de transferencia
   USB_EP1_RX_SIZE & 0xFF, USB_EP1_RX_SIZE >> 8,     //Tamaño máximo de paquete
   0x01,                    //  intervalo de lecturas en [ms] (Para transferencias
Isócronas)
};
```

Descriptor de cadena

```
char const USB_STRING_DESC[]={
     4, //tamaño del índice de cadena
     USB_DESC_STRING_TYPE,              //Tipo de descriptor 0x03 (STRING)
     0x09, 0x04,                 //Código para Inglés
  //Cadena 1 --> la compañía del producto
     8,                                 //Tamaño de cadena
     USB_DESC_STRING_TYPE,              //Constante para descriptor de cadena(0x03)
     'A', 0,
     'M', 0,
     'C', 0,
  //Cadena 2 --> Nombre del dispositivo
     22,                                //tamaño de la cadena
     USB_DESC_STRING_TYPE,              // Constante para descriptor de cadena(0x03)
     'U', 0,
     'N', 0,
     'A', 0,
     'M', 0,
     ' ', 0,
     'F', 0,
     '.', 0,
     'I', 0,
     '.', 0,
     ' ', 0
};
```

Con lo anterior, el dispositivo informará al equipo *Host* que nuestro dispositivo es *Full-Speed*, tiene dos *endpoints bulk*, uno de entrada y otro de salida, también informará el nombre del fabricante y nombre del dispositivo. Un aspecto importante a notar es que tanto el identificador de fabricante como el identificador de producto han sido seleccionados aleatoriamente, debido a que estos códigos deben ser proveídos por el USB-IF, si el producto va a ser comercializado, por lo que para el objetivo perseguido, esto no es necesario, estos valores denominados VID y PID, serán utilizados más adelante en el desarrollo del controlador para Windows, ya que el sistema operativo identifica a los dispositivos USB conectados al sistema leyendo su VID y PID, que deberán ser únicos para cada dispositivo.

Ahora bien, a continuación se mostrará la porción del código fuente que permite trabajar con el bus USB, pero antes es necesario recordar que para poder hacer cualquier transmisión primero es necesario que el dispositivo sea enumerado, y también es necesario recordar que todas las transmisiones sólo pueden ser realizadas, siempre y cuando el *Host* haga la petición por ellas, es decir el dispositivo no puede iniciar ninguna transmisión si no es requerida por el *Host*, como se encuentra definido en la especificación.

```
void main(void) {
  int8 recibe[3];
  int8 envia[1];

  output_low(PIN_B6);
  output_high(PIN_B7);
```

```
  printf("Inicializando\n");
  usb_init();           //inicializar el USB
  printf("Inicializacion terminada\n");

  usb_task();           //habilita USB e interrupciones
  usb_wait_for_enumeration();   //Esperar a ser configurado

  while (TRUE)
  {
   if(usb_enumerated())     //Si el dispositivo ha sido enumerado
   {
     if (usb_kbhit(1))     //Si el endpoint de salida contiene datos del host
     {
       usb_get_packet(1, recibe, 3);//recibir paquete en EP1 de entrada de tamaño 3
bytes
             //…
       usb_put_packet(1, envia, 1, USB_DTS_TOGGLE);
                                 //enviar un paquete de tamaño 1 por EP1 de salida
       //…
}}}
```

Este código muestra cómo inicializar el dispositivo USB para poder generar una transmisión por los *endpoints* declarados en los descriptores, cómo esperar la correcta enumeración del dispositivo y cómo recibir y enviar un paquete por los *endpoints* de entrada y salida, respectivamente.

En este apartado sólo se mostró una porción del código fuente, la versión completa se encuentra en el apéndice de este trabajo.

Con esto se termina el desarrollo del firmware para el dispositivo, es decir, en este momento se puede conectar al *Host*, reportar todas las características y funcionalidades, es decir, el dispositivo está terminado y totalmente funcional, en términos de hardware. Ahora bien, al conectarlo a un equipo, el sistema operativo requerirá de un controlador, ya que como se indicó en el descriptor de dispositivo éste no pertenece a ninguna clase predefinida sino que se declaró la clase definida por el fabricante, por lo que se requiere forzosamente de un controlador; a continuación se analizará el desarrollo de éste.

3.1.5 Desarrollo del controlador para Windows

El sistema operativo Windows, así como otros, soporta miles de dispositivos existentes. Se han liberado más de 30,000 controladores para estos dispositivos, y diariamente este número va creciendo. Algunos de estos controladores están basados en un modelo que fue diseñado hace más de 10 años, por lo que son obsoletos y deben ser modificados debido a sus limitaciones. Microsoft ha tomado establecido los pasos necesarios para simplificar el desarrollo de controladores para Windows, para mejorar su calidad y confiabilidad.

3.1.5.1 Kernel vs. usuario

Cuando se implementan aplicaciones, parece que el desarrollo de otro tipo de programas, tales como bibliotecas dinámicas o incluso controladores para el sistema operativo, son mundos totalmente distintos, pero al ir conociendo un poco de estas áreas, puede notarse que las tres son muy similares, todas tienen un punto de inicio **WinMain** para el caso de las aplicaciones, **DriverEntry** para el caso de los controladores y **DllMain** para las bibliotecas; se tienen funciones, variables, cabeceras, etc. Lo único en que realmente difieren es que se ejecutan en ámbitos totalmente distintos. Las aplicaciones y las bibliotecas (DLL), se desenvuelven en el ámbito denominado ***Modo Usuario,*** donde las aplicaciones del usuario son ejecutadas y todos los requerimientos de memoria y acceso a recursos del equipo son administrados y controlados por el sistema operativo, por otro lado los controladores se ejecutan en ***Modo Kernel***. El sistema operativo se ejecuta en este último donde se tiene permiso de acceder a la memoria, interrupciones de hardware, reloj, paginación, etc., objetos que serían imposibles de acceder desde modo usuario sin el uso de la API[20] para llamar a un *controlador*, que es la pieza de software que permite a las aplicaciones acceder de manera limitada a dispositivos tales como la impresora, el mouse, la tarjeta de video, los puertos USB, entre otros. La Figura 11 ilustra lo expuesto anteriormente.

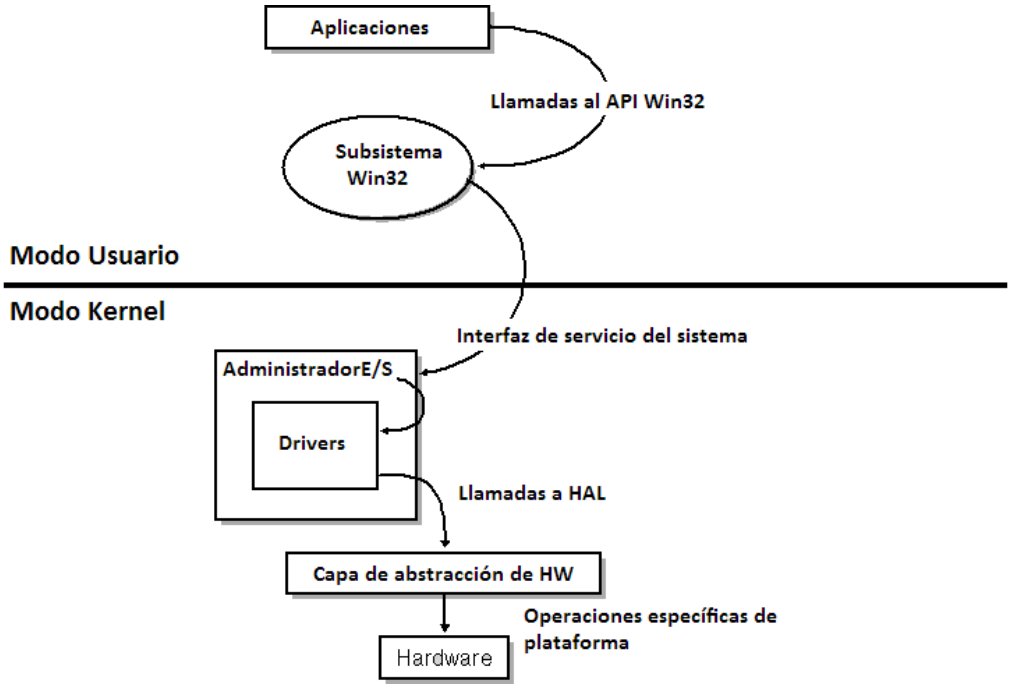


Figura 11. Esquema de modo kernel y modo usuario

[20] API, *Application Programming Interface*, Interfaz de programación de aplicación, es una interfaz de código fuente, que una aplicación, sistema operativo o biblioteca provee para soportar servicios requeridos por un programa de computadora.

3.1.5.2 Modelo actual

Windows actualmente soporta el modelo denominado *Windows Driver Model* (WDM) y muchos controladores específicos para clases de dispositivo tales como almacenamiento, redes y sonido. WDM provee una cantidad amplia de características, incluyendo transmisiones asíncronas, desarrollo en capas, *Plug and Play*, administración de energía, etc.

El modelo de los controladores específicos para dispositivos de clase, están típicamente estructurados en dos partes la *port* y la *miniport*. La parte *port* la escribe Microsoft que provee todas las funcionalidades generales sobre un estándar, y la versión *miniport* es desarrollada por el fabricante del producto e implementa las funcionalidades específicas de su dispositivo. En la actualidad existen ya más de diez clases.

3.1.5.3 Limitaciones del modelo actual

Aunque el modelo WDM y los controladores específicos para dispositivos de clase proveen herramientas para construir controladores poderosos y eficientes, el hardware y el software han evolucionado y presentan las siguientes limitaciones importantes:

-WDM es de bajo nivel y complejo. Escribir un driver requiere miles de líneas de código, la mayoría para funcionalidades comunes. Por ejemplo, si se desea desarrollar un controlador usando las interfaces que implementen en su totalidad *Plug and Play* y administración de energía, es necesario diseñar una máquina de estados que contiene al menos 100 posibles estados, y los controladores que son multifuncionales requieren de una mayor complejidad.

-Las interfaces de controlador de dispositivo no fueron diseñadas para el uso que se les da actualmente. Cuando se inició, Microsoft nunca previó que fabricantes diseñaran e implementaran controladores. Por lo que las interfaces de controlador fueron exportadas directamente del núcleo del sistema operativo, y por tanto cualquier controlador tiene la capacidad de acceder a estructuras esenciales para el sistema, lo que también impide que pudieran ser modificadas.

-La mayoría de los controladores debe ejecutarse en modo *kernel*, por lo tanto son tratados como parte del sistema operativo y tienen acceso a la memoria virtual. Es por esto que un error en un controlador provoca errores en el sistema operativo generando bloqueos en el sistema.

3.1.5.4 Windows Driver Foundation (WDF)

El modelo provisto en Noviembre de 2005, diseñado para solucionar los problemas anteriores tiene los siguientes componentes principales[21]:

- El *framework*[22] para controladores en modo Kernel (KMDF, *Kernel Mode Driver Framework*)

[21] http://www.microsoft.com/whdc/driver/wdf/KMDF-arch.mspx
[22] Se le conoce como *framework* al conjunto de bibliotecas diseñadas para resolver un problema complejo.

- El *framework* para controladores en modo usuario
- Herramientas para la verificación de controladores.

Este modelo está orientado a objetos y es controlado por eventos, por lo que agiliza el desarrollo de controladores, y permite el incremento de funcionalidades agregando poco código.

En otras palabras, este modelo está diseñado para que los desarrolladores de controladores, se concentren en las funcionalidades de su dispositivo y no en el sistema operativo.

Kernel Mode Driver Framework (KMDF)
Este modelo implementa las características fundamentales para los controladores en modo *kernel*, incluyendo soporte completo para *Plug and Play*, administración de energía, colas de E/S, DMA, transferencias tanto síncronas como asíncronas, además de contar con diseño específico para los buses USB y *Firewire*. Este esquema no se ejecuta en el núcleo del sistema operativo, sino que se ejecuta como una biblioteca separada.

User Mode Driver Framework (UMDF)
Este modelo implementa un pequeño grupo de funciones disponibles en KMDF, implementa *Plug and Play*, administración de energía y transmisiones Asíncronas. Los controladores que no requieren de acceso DMA, soporte a interrupciones, o que requieran de funciones de kernel, deberán ser desarrollados en este modo.

Con lo analizado anteriormente, el mejor esquema para desarrollar el controlador para el dispositivo en este trabajo es KMDF bajo WDF, ya que provee soporte inherente para el bus USB sobre el que se está trabajando.

3.1.5.5 Desarrollo del controlador con KMDF

El desarrollo del controlador es inherente al *framework* ya que KMDF fue diseñado con el puerto USB en mente; provee las funciones y macros dedicadas exclusivamente al control y administración de dispositivos, interfaces, *endpoints*, y las transferencias de todos los tipos. Además provee implementaciones por defecto para administración de energía y *Plug and Play*, por lo que el desarrollo del controlador es rápido. A continuación se muestra un fragmento del código fuente para observar cómo se implementan los *endpoints* requeridos.

```
NTSTATUS ConfigureUsbPipes(PDEVICE_CONTEXT DeviceContext)
{
 NTSTATUS status = STATUS_SUCCESS;
 BYTE index = 0;
 WDF_USB_PIPE_INFORMATION pipeConfig;
 WDFUSBPIPE pipe = NULL;

 DeviceContext->UsbBulkInPipe = NULL;
 DeviceContext->UsbBulkOutPipe = NULL;
```

```
 WDF_USB_PIPE_INFORMATION_INIT(&pipeConfig);



 do
 {
  pipe = WdfUsbInterfaceGetConfiguredPipe(DeviceContext->UsbInterface
                     index,//Índice del pipe
                     &pipeConfig);//datos leidos del pipe[index]
  if(NULL == pipe)
   break;//repetir mientras haya pipes válidos

  /*Ninguna transferencia asegurará un tamaño fijo de transferencia de datos.*/
  WdfUsbTargetPipeSetNoMaximumPacketSizeCheck(pipe);


  if(WdfUsbPipeTypeBulk == pipeConfig.PipeType)
  {
   if(TRUE == WdfUsbTargetPipeIsInEndpoint(pipe))
   {
    DeviceContext->UsbBulkInPipe = pipe;
   }
   else if(TRUE == WdfUsbTargetPipeIsOutEndpoint(pipe))
   {
    DeviceContext->UsbBulkOutPipe = pipe;
   }
  }
       index++;
 } while(NULL != pipe);

 if( (NULL == DeviceContext->UsbBulkInPipe) ||
        (NULL == DeviceContext->UsbBulkOutPipe))
 {
  KdPrint((__DRIVER_NAME
   "No se encontraron todos los pipes.\n"));
  return STATUS_INVALID_PARAMETER;
 }

 return status;
}
```

Como puede observarse en el código anterior, se verifica que existan los dos *endpoints* requeridos en este trabajo. Se le llaman *pipes* a las conexiones lógicas, es decir a los *endpoints* en la interfaz, también es de notarse que las estructuras utilizadas son de diseño específico para el protocolo USB. El código completo del controlador se encuentra en el apéndice de este trabajo.

Para la compilación del controlador es requerido el uso del kit para desarrollo de controladores, denominado DDK, con la llegada de Windows Vista y el WDF, ha cambiado de nombre a WDK, kit de controladores para Windows, y como se mencionó es requerida la instalación de KMDF para el kit. El

controlador desarrollado en este trabajo es compatible tanto con Windows XP y Windows Server 2003, como con Windows Vista[23].

3.1.5.6 Desarrollo de la aplicación para Windows

Para el desarrollo de esta aplicación en modo usuario, es necesario implementarla con las funciones del API de Windows, *CreateFile, ReadFile, WriteFile*. Para Windows cualquier cosa que sea susceptible a escritura o lectura, se maneja como archivo y de esta manera se simplifican muchas actividades, el caso de escrituras y lecturas al puerto USB no es la excepción, por lo que para poder comunicarse con el dispositivo USB desarrollado hasta el momento es necesario primero crear el "archivo", una vez creado se devolverá un puntero a este, con el cual será posible tanto leer como escribir. Ahora bien para poder conocer el nombre del archivo es necesario conocer cómo identifica Windows a nuestro dispositivo una vez que ha sido conectado y enumerado. Esto se hace leyendo en el registro del sistema operativo y buscando el GUID[24], que es un identificador único para dispositivo y con el que será instalado el controlador desarrollado anteriormente. Esta información se encuentra en el registro HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceClasses y la subllave tendrá como nombre el GUID, encerrado entre corchetes. Ahora bien, la manera de implementar lo anterior debe ser en C o en C++, por lo que para facilitar el desarrollo de este proyecto y simplificar el modo de uso, la aplicación para acceder a las comunicaciones con el dispositivo USB, será desarrollada en forma de Librería dinámica (DLL), lo que permitirá el uso de cualquier lenguaje de programación que permita el ligado de librerías en tiempo de ejecución, de esta manera se proveerá de un modo fácil, sencillo y seguro de utilizar el dispositivo USB, es decir una interfaz de programación. La implementación de la librería fue desarrollada en Microsoft Visual C++, el código completo se encuentra en el apéndice.

[23] Debe ser deshabilitada la verificación de firma digital para los controladores.

[24] GUID, *Globally Unique IDentifier*, es un tipo especial de identificador usado en aplicaciones de software para proveer un número único en el contexto donde se utiliza la aplicación, puede proveer hasta 2^{122} números distintos por lo que es muy poco probable que se genere el mismo número dos veces.

3.2 Puerto paralelo

La implementación del puerto paralelo es muy sencilla, ya que tanto la PC como el microcontrolador tienen interfaz paralela, por lo que sólo es necesario enviar los datos por el puerto para que el microcontrolador reciba los datos y reaccione según sea programado; el único inconveniente es que el puerto paralelo en modo SPP no soporta lectura de datos por el puerto de datos, por lo que deberá agregarse un multiplexor para que se puedan leer los 8 bits, pero esto permitirá una conexión *Full dúplex*.

3.2.1 Diseño electrónico

El microcontrolador deberá ser conectado como se muestra en la Figura 12 para poder enviar y recibir datos.

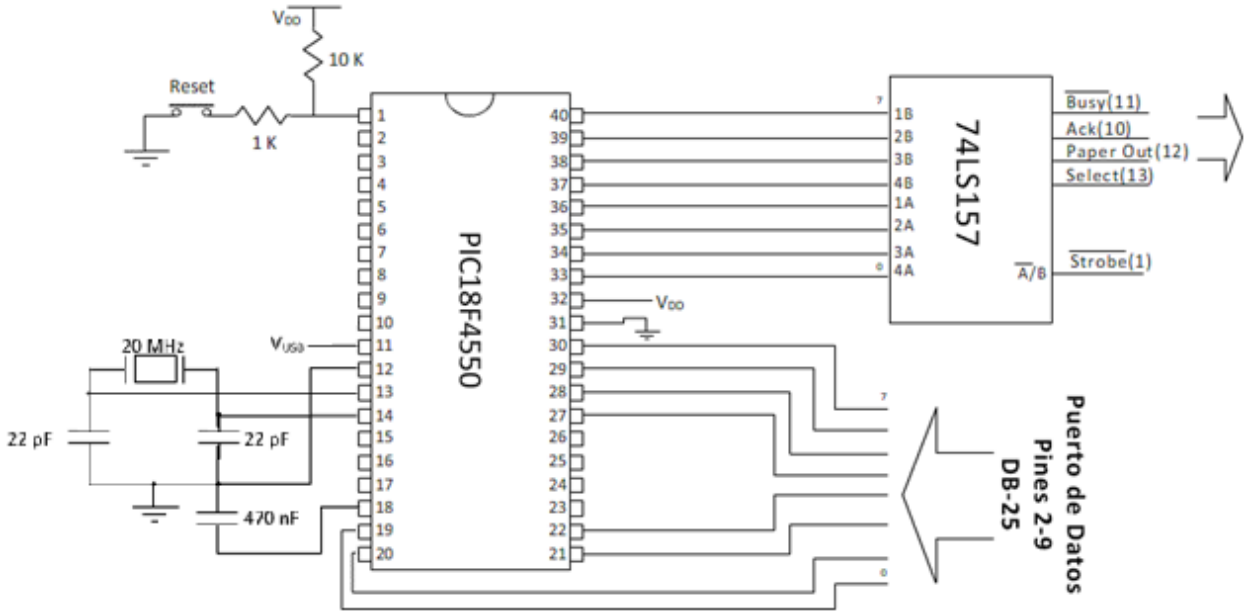


Figura 12. Circuito para la implementación del puerto paralelo con comunicación *Full-Duplex*

Este circuito utiliza el multiplexor 74LS157 que tiene doble entrada de cuatro bits y una salida de *nibble*[25], por lo que sólo se necesita una señal (en este caso *Strobe*), para hacer el intercambio de *nibbles*.

Otra posible solución a este problema es utilizar el puerto paralelo SPP en modo bidireccional, pero debido a que todos los puertos paralelos en el mercado no son bidireccionales, ni tampoco pueden ser configurados de la misma manera, por lo que es mejor utilizar el modo nibble, que es estándar a todos los puertos SPP existentes.

[25] Se le llama *nibble* al conjunto de 4 bits altos o bajos dentro de un byte.

3.2.2 Desarrollo del firmware

Para este firmware el programa es muy sencillo, ya que el microcontrolador trabaja los puertos como paralelos, sólo es necesario configurarlos correctamente si son de entrada o de salida. A continuación se muestra un segmento del código fuente.

```
#include<18f4550.h>
#use delay(clock=20000000)

//tris : 1:input, 0: output
void main(){
 //Configurar puertos:
 set_tris_b(0x00);  //PORTB: Output
 set_tris_d(0xFF);  //PORTD: Input

 while(1){//repetir siempre
    //...
    output_b(DATO); //salida de datos por puerto B
    //...
    DATO=input_d(); //lectura de datos por puerto D
    //...
 }
}
```

De esta manera se muestra cómo se hacen las lecturas y escrituras en los puertos en el microcontrolador, aunque es necesario hacer la aplicación que haga las lecturas y escrituras desde y hacia el puerto paralelo en la PC.

3.2.3 Desarrollo aplicación para la PC

Para la aplicación del puerto paralelo, sólo es necesario recordar el hardware, las líneas a las que se conectó el circuito y cómo deben ser interpretadas.

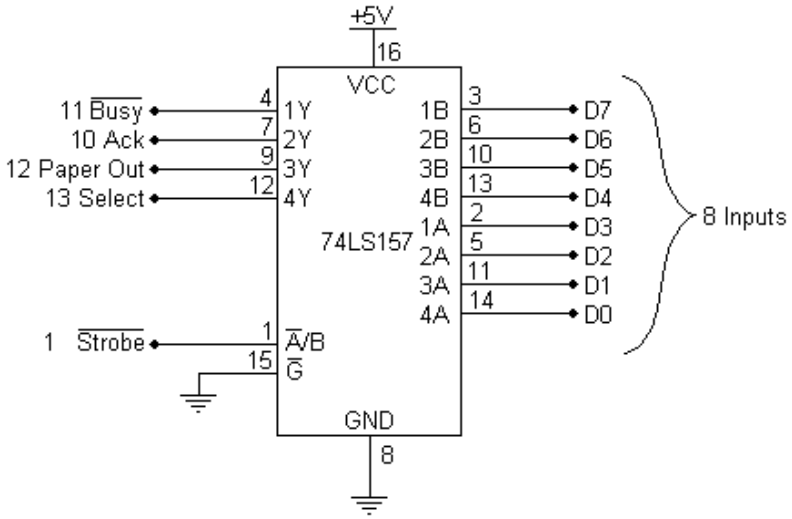


Figura 13. Forma de conectar el puerto paralelo en modo *nibble*

El multiplexor funciona de la siguiente manera, cuando la línea de selección de *nibble* está en bajo lógico, las entradas A son seleccionadas, es decir A pasa al registro de estado y cuando la línea de selección de *nibble* está en alto, las entradas en B son seleccionadas.

Para usar este circuito es necesario primero inicializar el multiplexor para seleccionar A o B, por lo tanto para leer el *nibble* bajo, se debe establecer en bajo la línea A/B, y como este *pin* está conectado a la línea *Strobe* del puerto, que es una línea invertida, deberá ponerse en alto el Bit C0 del registro de control, para conseguir el bajo en el pin 1 del puerto paralelo. Lo anterior se logra escribiendo en el registro de control un 0x01, para así seleccionar el *nibble* menos significativo. Una vez obtenido el dato, es necesario recorrerlo cuatro bits a la derecha, ya que se está leyendo el *nibble* menos significativo, y se está obteniendo el dato del registro de estado que tiene las líneas conectadas en el *nibble* más significativo. Ahora se deberá seleccionar el *nibble* más significativo, se logra estableciendo un cero en el Bit C0 del registro de control, con esto puedo leerse nuevamente el dato faltante. Por último, es necesario recordar que como se está usando la línea *Busy*, que es invertida, el dato que se leyó en ambos *nibbles* estará también invertido por lo que es necesario aplicar una función XOR con la máscara 0x88, para reinvertir los bits afectados. El código fuente para hacer lo anterior es el siguiente:

```
outportb(CONTROL, inportb(CONTROL) | 0x01);    /* Seleccionar el nibble bajo (A) */
a = (inportb(STATUS) & 0xF0);                  /* Leer el nibble bajo            */
a = a >> 4;                                    /* Recorrer 4 bits a la derecha   */
outportb(CONTROL, inportb(CONTROL) & 0xFE);    /* Seleccionar nibble alto (B)    */
a = a | (inportb(STATUS) & 0xF0);              /* Leer nibble alto               */
a = a ^ 0x88;                                  /* Reinvertir los bits            */
```

3.3 Puerto serial

Para el desarrollo de esta sección, se utilizó el UART que viene integrado en el microcontrolador, por lo que el desarrollo es directo y sencillo y permitiendo velocidades de hasta 115,200 bps.

3.3.1 Diseño electrónico

Para este circuito también es necesario un circuito extra para la comunicación serial, como se comentó anteriormente en la sección 2.1.3. En la figura 14 se muestra el diagrama del circuito.

De igual manera, este circuito es capaz de enviar y recibir datos seriales en modo Full-dúplex.

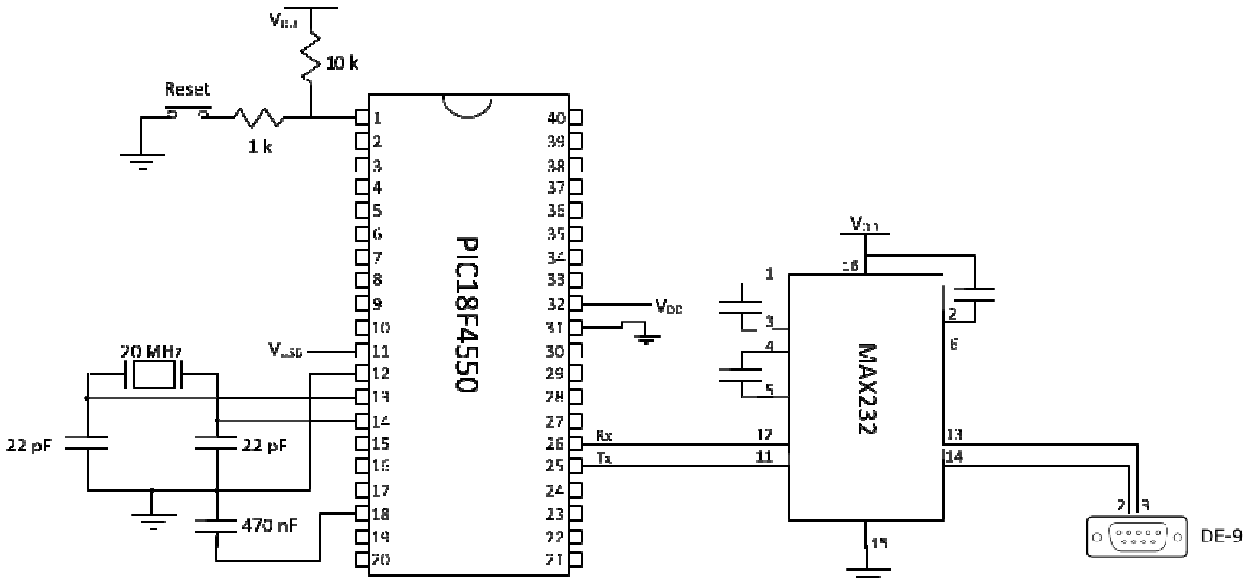


Figura 14. Circuito para la implementación del puerto serial

3.3.2 Desarrollo de firmware

El PIC trae integrado el UART para comunicación serial, por lo que la implementación se realiza de manera sencilla y rápida. El siguiente programa muestra cómo hacer transmisiones y recepciones por el puerto serial.

```
#include<18f4550.h>
#use delay(clock=20000000)
#use rs232(baud=19200,xmit=PIN_C6,rcv=PIN_C7) //establecer velocidad y definir pines

//tris : 1:input, 0: output
void main(){

 while(1){//repetir siempre
    //...
    DATO=getc();    //obtiene un dato de 8 bits (un carácter)
    //...
    DATO[]=gets();  //obtiene una cadena de caracteres hasta recibir \r
    //...
    putc(DATO);           //escribe un byte en el puerto serie
    //...
    puts(DATO[]);   //escribe una cadena de caracteres o bytes en COM
    //...
    printf("%d",DATO);     //escribe una cadena con formato en el puerto serie
 }
}
```

Como puede observarse, la transmisión de datos vía serial es sencilla, y el programa puede dedicarse a otras actividades tales como lectura de canales de conversión analógica a digital, control de motores o actuadores.

3.3.3 Desarrollo de la aplicación para Windows

El desarrollo de esta aplicación fue realizado en lenguaje C# debido a que, en conjunto con el *framework* .NET, provee interfaces de programación para comunicarse directamente al puerto serial y además facilita el desarrollo de aplicaciones que requieran recepción controlada por eventos, es decir, que la aplicación responda a eventos de recepción, sin tener la aplicación bloqueada en un ciclo *while* para esperar la llegada de datos (*polling*).

3.4 Desarrollo de la Aplicación de Internet

La aplicación de Internet es aquella que se encargará de proveer una interfaz gráfica para el usuario, recibir todas las peticiones y llamar a la aplicación correspondiente, dependiendo del modo de comunicación seleccionado. Este desarrollo se realiza utilizando Microsoft Visual Studio.

El desarrollo de esta aplicación fue implementado en lenguaje C# sobre el *framework* .NET, que provee soporte tanto para aplicaciones Windows como aplicaciones y servicios web, ejecutándose sobre IIS(*Internet Information Services*), que es un servidor de Internet que se encarga de recibir las peticiones de despliegue de páginas en Internet, ejecutando aplicaciones en el servidor, sin que el usuario tenga acceso a éstas, y mostrando los resultados en HTML plano, es decir, en HTML que puede ser desplegado por cualquier explorador de internet, al cliente que se encuentra conectado.

El siguiente código es un ejemplo muy sencillo de una aplicación que se ejecuta en el servidor:

```
<script runat="server">
       void submit(object sender, EventArgs e)
       if (name.value!="")
         p1.InnerHtml="Bienvenido " + name.value + "!"
       end if
       End Sub
</script>

<html>
<body>

       <form runat="server">
       Pon tu nombre: <input id="name" type="text" size="30" runat="server" />
       <br /><br />
       <input type="submit" value="Enviar" OnServerClick="submit" runat="server" />
       <p id="p1" runat="server" />
       </form>

</body>
</html>
```

Como puede observarse, la anterior aplicación se encuentra insertada sobre un archivo HTML, pero la extensión de estos archivos no es *.htm* o *.html* sino ***.aspx** (Active Server Pages)*, que indica que es una aplicación que se ejecuta sobre el *Framework* .NET, además puede notarse que los *tags* (etiquetas) que

forman parte de la aplicación tienen un parámetro especial que dice *runat="server"* (Ejecución en el servidor).

En esta pequeña aplicación, IIS genera un HTML que muestra al usuario un cuadro de texto que indica "Pon tu nombre", delante de este se mostrará un botón que dice "Enviar"; si el usuario presiona el botón Enviar, la aplicación ejecutará la función mostrada arriba, y hará la verificación de que se haya escrito algo y, si así es, entonces debajo del cuadro de texto mostrará "Bienvenido [texto que se haya escrito] ".

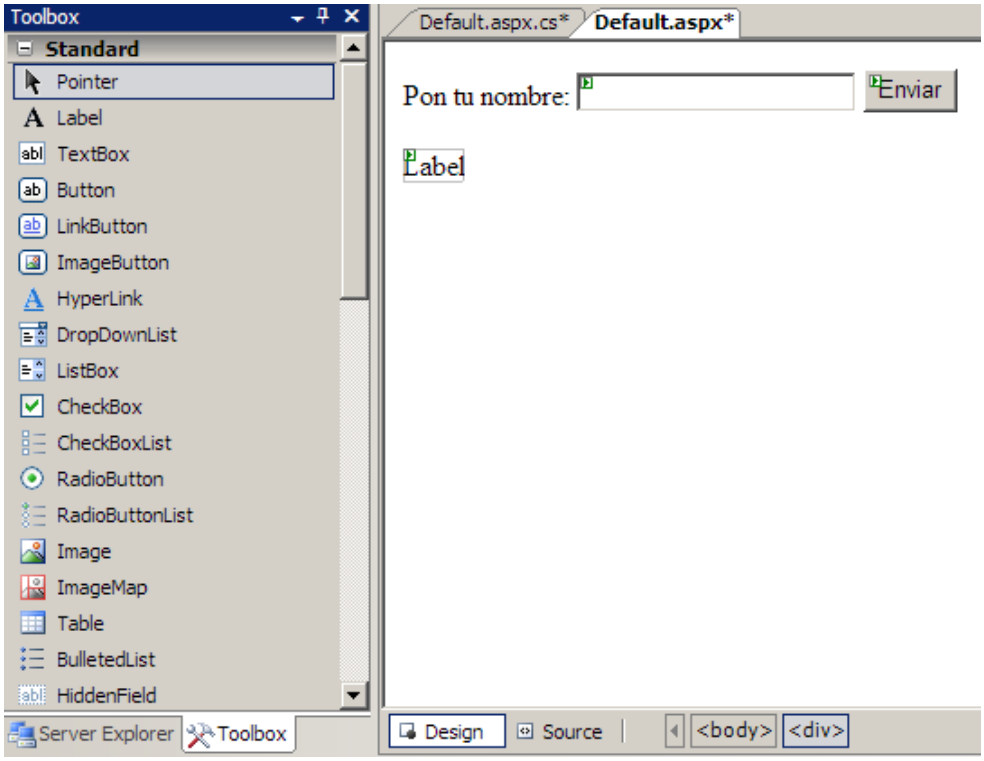


Figura 15. Entorno de desarrollo de Visual Studio para páginas dinámicas

Microsoft Visual Studio, provee una manera sencilla y cómoda se hacer todo lo anterior, permitiendo dibujar la página web tal y como deseamos que sea mostrada (lo que se conoce como "*What you see is what you get*", WYSIWYG), facilitando la inserción de objetos, seleccionándolos de una barra de herramientas e insertando automáticamente el código para cada objeto, acelerando así el desarrollo de aplicaciones, tal y como se muestra en la Figura 15.

Aquí es posible ver la misma aplicación, que sólo contenía código fuente, implementada sólo arrastrando los objetos deseados en el espacio de trabajo.

Puede notarse que algunos objetos tienen un pequeño triángulo verde que indica que estos objetos son objetos dinámicos, es decir pueden ser programados. Para agregar la acción, es necesario sólo dar doble

clic sobre el botón enviar e insertar el código que verifique que se haya escrito algo y desplegar "Bienvenido [texto escrito]".

```
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void Button1_Click(object sender, EventArgs e)
    {
        if (TextBox1.Text != "")
            Label1.Text = "Bienvenido " + TextBox1.Text;
    }
}
```

Figura 16. Código fuente en el entorno de desarrollo de Visual Studio

En el capítulo siguiente se analizará cómo conjuntar el servicio de Internet con las otras aplicaciones.

Capítulo 4

Control remoto de interfaces

El objetivo central de este trabajo es el de controlar remotamente el equipo que se encuentra en el laboratorio, por lo que en este capítulo, se analizará la forma en que todos los componentes implementados se integran para lograr la comunicación usuario laboratorio. La manera de diseñar el control remoto es la siguiente

4.1 Ejecución remota de aplicaciones

El servicio web propiamente no tiene permisos para poder acceder a los puertos, pero sí tiene permisos para ejecutar otras aplicaciones. Las aplicaciones llamadas pueden tener acceso a los puertos, pero deben ser diseñadas de manera que no requieran interactividad, por lo que las aplicaciones en modo consola forman un candidato potencial, ya que pueden sólo desplegar información, si se programan de manera que no requieran entrada del usuario, y además tienen la capacidad de tener datos de entrada, en este caso los parámetros que permiten definir el comportamiento de la aplicación.

Como se analizó en el capítulo anterior, el desarrollo de aplicaciones web es sencillo, y ayudándose del *framework* .NET es posible tener acceso a muchas funciones ya programadas. En este caso el *framework* .NET para aplicaciones web contiene un conjunto de herramientas para poder realizar lo anteriormente planteado, es posible llamar a otras aplicaciones para que sean ejecutadas en el mismo servidor de Internet y, de esta manera el cliente podrá ejecutar procesos en el equipo remoto, es decir, para este trabajo, los alumnos que se conecten desde su equipo, la interfaz gráfica irá ejecutando aplicaciones que controlan los puertos en el servidor al que se conectan, y a su vez estas aplicaciones controlarán los equipos en el laboratorio vinculados al mismo servidor, todo esto de manera transparente al alumno que se encuentra realizando las prácticas.

Otro aspecto importante es que el objeto *Process*, que provee el framework permite redireccionar tanto la salida estándar como la entrada estándar, es decir, que si se requiriera en algún caso cierta interactividad, sería posible utilizando estas propiedades. De esta manera, también es posible mostrar los resultados de la lectura, directamente en la página web generada.

4.2 Aplicaciones para control remoto

Ahora bien, las aplicaciones para que los puertos sean controlados mediante línea de comandos y los argumentos necesarios para definir el control de las interfaces, sería el siguiente:

Puerto serial

Sólo es necesario definir si se hará una lectura o una escritura, ya que para el caso de este puerto su dirección es siempre es la misma y la configuración siempre será constante, por lo tanto la sintaxis será la siguiente:

```
\>PuertoSerie.exe [Sin Argumentos (Lectura)] [Datos (escritura)] [-f (Toma datos como el nombre de un archivo de texto)]
```

Si sólo se provee el modificador **–f** almacenará los datos leídos en el puerto en un archivo nombrado ***PuertoSerie.txt***, en el mismo directorio donde se encuentra el ejecutable.

Puerto paralelo

En este caso, es necesario definir a qué puerto se desea escribir o leer, ya que dependiendo del registro al que se desee acceder, la dirección del puerto será distinta.

```
\>ParPort.exe [Dirección] [Datos a enviar] [-f (Toma datos como el nombre de un archivo de texto o si es lectura crea un archivo con los resultados)]
```

Si sólo se provee la dirección, se tomará como una operación de lectura y mostrará el resultado en la salida estándar.

Si sólo se provee la dirección y el modificador **–f**, almacenará los datos leídos en un archivo llamado ***ParPort.txt*** en el mismo directorio en el que se encuentra ubicado el ejecutable.

Puerto USB

Para este puerto, será necesario indicar si es una operación de lectura o escritura y, la aplicación seleccionará automáticamente el *endpoint* apropiado.

```
\>USBPort.exe [Sin argumentos (lectura)] [Datos a enviar] [-f (Toma datos como el nombre de un archivo de texto o si es lectura crea un archivo con los resultados)]
```

Si se utiliza sin argumentos, se mostrarán los resultados en la salida estándar; si se agrega **–f** se creará un archivo llamado ***USBPort.txt*** en el mismo directorio donde se ubica la aplicación con los datos leídos del *endpoint*.

4.3 Integración de aplicaciones con el control remoto

Ahora bien, como se comentó líneas arriba, el *framework* provee los métodos necesarios para lograr la ejecución remota de aplicaciones; el siguiente fragmento de código fuente, muestra cómo ejecutar una aplicación en el equipo donde se encuentra ejecutándose el servidor web.

```
Process myProcess = new Process();
ProcessStartInfo info = new ProcessStartInfo("aplicacion.exe");
myProcess.StartInfo = info;
myProcess.Start();
myProcess.Close();
```

De esta manera se indica al servidor que cree un nuevo proceso con el archivo "aplicacion.exe", para que inicie su ejecución. Ahora bien, este fragmento sólo ejecuta y termina, pero es necesario redirigir la salida estándar, es decir, para que no sea enviada a la consola sino a donde se desee. Por lo que es necesario primero deshabilitar la consola, una vez hecho esto, es posible asignar la salida estándar a una variable o a la misma página web; lo anterior se logra modificando la propiedad ReDirectStandardOutPut, que se hace de la siguiente manera.

```
   Process myProcess = new Process();
myProcess.StartInfo.UseShellExecute = false;
   myProcess.StartInfo.RedirectStandardOutput = true;
   myProcess.Startinfo.FileName = "aplicacion.exe";
myProcess.Start();
   string output = myProcess.StandardOutput.ReadToEnd();
myProcess.WaitForExit();
```

Ahora bien, este código corrige dos situaciones, la primera, planteada anteriormente, el redireccionamiento de la salida estándar, en este caso a una variable denominada *output*. El otro problema que no es observable a simple vista, es que si no se hubiera usado el método `ReadToEnd` en `StandardOutput`, que es una lectura asíncrona, sino sólo asignar a *output* la salida estándar, podría provocar problemas de bloqueo de sistema, ya que se está haciendo una lectura síncrona, es decir, la aplicación no podrá continuar sino hasta que `myProcess` termine de escribir en la salida estándar, pero si la aplicación falla, `myProcess` quedaría detenido indefinidamente esperando a que termine de escribir, provocando un bloqueo en la aplicación web.

Hasta este momento sólo falta resolver el envío de argumentos a la aplicación a ejecutar, y con esto se tendría terminado el control remoto. Los argumentos están asignados a una variable en el objeto `ProcessStartinfo`, llamada `Arguments`, que es de tipo cadena, por lo que sólo sería necesario asignar los argumentos necesarios a la variable, antes de iniciar el proceso.

A continuación se muestra un ejemplo para la aplicación que envía un dato al puerto paralelo.

```
   Process myProcess = new Process();
myProcess.StartInfo.UseShellExecute = false;
   myProcess.StartInfo.RedirectStandardOutput = true;
   myProcess.Startinfo.FileName = "ParPort.exe";
   myProcess.Startinfo.Arguments = "888 255";
myProcess.Start();
   string output = myProcess.StandardOutput.ReadToEnd();
myProcess.WaitForExit();
```

De esta manera queda implementado totalmente el control remoto de puertos.

Capítulo 5

Resultados y Conclusiones

El proyecto logró cumplir en su totalidad con los requerimientos, sin embargo al ejecutarse sobre una red como lo es Internet, el control remoto se encuentra sometido a muchas variables que pueden menguar el rendimiento del sistema en general, ya que el tránsito de Internet, a pesar de viajar sobre protocolos tan robustos como lo es TCP, aún es posible encontrar retrasos en el envío de paquetes, lo que podría provocar *timeouts*[26] indeseados en la aplicación web; se han tomado en cuenta estos posibles problemas, y se recomienda el incremento de estos tiempos en las configuraciones del servidor de Internet, para evitar al máximo estos errores que son parte inherente a cualquier servicio de Internet, para así asegurar el máximo tiempo de disponibilidad del sistema.

Otro aspecto limitante y que puede ser modificado fácilmente, es que en el dispositivo USB sólo se cuenta con dos *endpoints*, que aunque son suficientes para este proyecto, podrían no serlo para otras aplicaciones; para resolver esto tendrían que modificarse tanto los descriptores como el controlador de dispositivo para Windows, para abordar estos cambios.

A pesar de los problemas anteriores, se puede asegurar que el diseño tanto de hardware como de software son lo suficientemente robustos como para trabajar sin descanso, y con mantenimiento mínimo, ya que se han realizado las pruebas necesarias para verificar el continuo funcionamiento de los dispositivos.

La forma en que fue desarrollado este trabajo permite la utilización individual de las interfaces para otras aplicaciones, ya que fueron implementadas de manera muy general, y por tanto es fácilmente aplicable a otras áreas de la ingeniería.

Para mejorar el sistema de interfaz a futuro, es necesario que se implemente el puerto paralelo en modo bidireccional para tener comunicación *Full-Dúplex;* para lograr lo anterior se recomienda el uso del modo EPP del puerto paralelo, ya que este modo de funcionamiento se implementa de manera sencilla y a diferencia del modo ECP, aquél genera y controla todas las transferencias hacia y desde el periférico. Por otro lado para optimar la interfaz USB, es necesario utilizar la versión 2.0 del protocolo,

[26] Se le llama *timeout* a la cantidad de tiempo preestablecido para esperar a que un sistema responda, si este tiempo es alcanzado se dispara un error, en el que se notifica que se ha dejado de esperar a que el sistema responda y de esta manera se evita esperar indefinidamente por una respuesta.

ya que ofrece velocidades de transferencia muy superiores y que en el futuro podrían llegar a ser requeridas; este cambio requeriría el uso de otro microcontrolador, ya que el usado en este trabajo no soporta el modo *High-Speed,* el costo del proyecto aumentaría y requeriría volver a generar el firmware del microcontrolador; el controlador para Windows no requeriría de ningún cambio, ya que está desarrollado de manera general y puede ser utilizado con cualquier dispositivo USB.

Por último, el producto final desarrollado es dinámico y en su configuración básica permite crear dispositivos usando un modo denominado HID (*Human Interface Device*, o dispositivo de interfaz humana), el cual no requiere un controlador para Windows lo cual facilitaría el transporte a otros equipos sin la necesidad de instalar el controlador en el equipo, el único inconveniente es que sólo podrían usarse hasta 1.5 Mbps de transferencia. Este trabajo también permite el uso de controladores de clase que facilitan al usuario el empleo del equipo en el sistema operativo, esta tendencia ha ido creciendo en la industria aprovechando al máximo la tecnología *Plug And Play*, ya que como se observó en el desarrollo de este trabajo hacer un controlador requiere de habilidad programática, comprender conceptos del sistema operativo y además es necesario contar con las herramientas necesarias, por lo que otra manera de mejorar el proyecto sería modificando los descriptores de clase del dispositivo para que se usen controladores de clase en lugar de un controlador definido por fabricante y para comunicarse con el dispositivo de Hardware sólo es necesario lidiar con el controlador de clase provisto con el sistema operativo.

Bibliografía

[1] **EIA/TIA Electronics industries Associations**
"Interface between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange ANSI/TIA-232-F-1997 (R2002)", septiembre 1997.
Sin ISBN

[2] **AXELSON, Jan**
"Serial Port Complete, Programming and Circuits for RS-232 and RS-485 Links and Networks", Lake View Research, 2000.
ISBN 0965081974

[3] **HYDE, John**
"USB Design by Example", Intel Press, 2001.
ISBN 0970284655

[4] **USB-IF**
*"Universal Serial Bus Revision 2.0 specification", a*bril 2000.
Sin ISBN

[5] **MCDONALD, Matthew**
"Microsoft® .NET Distributed Applications: Integrating XML Web Services and .NET Remoting", Microsoft Press, 2003.
ISBN 0735619336

[6] **ONEY, Walter**
"Programming the Microsoft Windows Driver Model 2nd Edition", Microsoft Press, 2003.
ISBN 0735618038

Mesografía

[1] http://beyondlogic.org/usbnutshell/usb1.htm
USB in a Nutshell - Making Sense of the USB Standard
Consultada en junio 2007 (en línea).

[2] http://www.fapo.com/porthist.htm
Warp Nine Engineering- Parallel Port Background
Consultada en junio 2007 (en línea).

[3] http://www.captain.at/serial-port-registers.php
Serial (RS232) Port Pin and Registers
Consultada en junio 2007 (en línea).

[4] http://zone.ni.com/devzone/cda/tut/p/id/3466
National instruments – IEEE1284 Updating the Parallel Port
Consultada en junio 2007 (en línea).

Capítulo 6

Apéndice

6.1 Código Fuente para el Microcontrolador PIC18F4550, que implementa la comunicación USB

6.1.1 Archivo de cabecera PicUSB.h

```
////////////////////////////////////////////////////////////////////////
////                          PicUSB.h                              ////
////                                                                ////
//// This examples shows how to configure the USB device and its    ////
//// descriptors, and how to send and receive data from the PC wait- ////
//// first for the correct enumeration.                             ////
////                                                                ////
//// Compiled using CCS PCWH 4.018                                  ////
////                                                                ////
//// Author: Abraham Monrroy Cano     abraham_monrroy@hotmail.com   ////
////                                                                ////
////////////////////////////////////////////////////////////////////////

#IFNDEF __USB_DESCRIPTORS__
#DEFINE __USB_DESCRIPTORS__

#include <usb.h>

//////////////////////////////////////////////////////////////////
///
///   start config descriptor
///   right now we only support one configuration descriptor.
///   the config, interface, class, and endpoint goes into this array.
///
//////////////////////////////////////////////////////////////////

   #DEFINE USB_TOTAL_CONFIG_LEN      32 //config+interface+class+endpoint

   //configuration descriptor
   char const USB_CONFIG_DESC[] = {
   //config_descriptor for config index 1
         USB_DESC_CONFIG_LEN,     //length of descriptor size
         USB_DESC_CONFIG_TYPE,    //constant CONFIGURATION (0x02)
         USB_TOTAL_CONFIG_LEN,0,  //size of all data returned for this config
         1,                   //number of interfaces this device supports
         0x01,                //identifier for this configuration.  (IF we had more than one
configurations)
         0x00,                //index of string descriptor for this configuration
         0xC0,                //bit 6=1 if self powered, bit 5=1 if supports remote wakeup (we
don't), bits 0-4 reserved and bit7=1
         0x32,                //maximum bus power required (maximum milliamperes/2)  (0x32 =
100mA)

   //interface descriptor 0 alt 0
         USB_DESC_INTERFACE_LEN,      //length of descriptor
         USB_DESC_INTERFACE_TYPE,     //constant INTERFACE (0x04)
         0x00,                        //number defining this interface (IF we had more than one
interface)
         0x00,                //alternate setting
         2,                   //number of endpoints, not counting endpoint 0.
         0xFF,                //class code, FF = vendor defined
         0xFF,                //subclass code, FF = vendor
         0xFF,                //protocol code, FF = vendor
         0x00,                //index of string descriptor for interface
```

```
   //endpoint descriptor
         USB_DESC_ENDPOINT_LEN,       //length of descriptor
         USB_DESC_ENDPOINT_TYPE,      //constant ENDPOINT (0x05)
         0x81,                        //endpoint number and direction (0x81 = EP1 IN)
         0x02, //transfer type supported (0 is control, 1 is iso, 2 is bulk, 3 is interrupt)
         USB_EP1_TX_SIZE & 0xFF,USB_EP1_TX_SIZE >> 8,  //maximum packet size supported
//USB_EP1_TX_SIZE,0x00,         //maximum packet size supported
         0x01,              //polling interval in ms. (for interrupt transfers ONLY)

   //endpoint descriptor
         USB_DESC_ENDPOINT_LEN,       //length of descriptor
         USB_DESC_ENDPOINT_TYPE,      //constant ENDPOINT (0x05)
         0x01,                        //endpoint number and direction (0x01 = EP1 OUT)
         0x02, //transfer type supported (0 is control, 1 is iso, 2 is bulk, 3 is interrupt)
         USB_EP1_RX_SIZE & 0xFF,USB_EP1_RX_SIZE >> 8,       //maximum packet size supported
//USB_EP1_RX_SIZE,0x00,       //maximum packet size supported
         0x01,               //polling interval in ms. (for interrupt transfers ONLY)

  };

   //****** BEGIN CONFIG DESCRIPTOR LOOKUP TABLES ********
   //since we can't make pointers to constants in certain pic16s, this is an offset table to find
   //  a specific descriptor in the above table.

   //NOTE: DO TO A LIMITATION OF THE CCS CODE, ALL HID INTERFACES MUST START AT 0 AND BE
SEQUENTIAL
   //      FOR EXAMPLE, IF YOU HAVE 2 HID INTERFACES THEY MUST BE INTERFACE 0 AND INTERFACE 1
   #define USB_NUM_HID_INTERFACES   0

   //the maximum number of interfaces seen on any config
   //for example, if config 1 has 1 interface and config 2 has 2 interfaces you must define this
as 2
   #define USB_MAX_NUM_INTERFACES   1

   //define how many interfaces there are per config.  [0] is the first config, etc.
   const char USB_NUM_INTERFACES[USB_NUM_CONFIGURATIONS]={1};

   #if (sizeof(USB_CONFIG_DESC) != USB_TOTAL_CONFIG_LEN)
      #error USB_TOTAL_CONFIG_LEN not defined correctly
   #endif


////////////////////////////////////////////////////////////////////
///
///   start device descriptors
///
////////////////////////////////////////////////////////////////////

   //device descriptor
   char const USB_DEVICE_DESC[] ={
         USB_DESC_DEVICE_LEN, //the length of this report
         0x01,                //constant DEVICE (0x01)
         0x10,0x01,           //usb version in bcd
         0x00,                //class code (if 0, interface defines class.  FF is vendor defined)
         0x00,                //subclass code
         0x00,                //protocol code
         USB_MAX_EP0_PACKET_LENGTH,   //max packet size for endpoint 0. (SLOW SPEED SPECIFIES 8)
         0x3f,0x44,           //vendor id (0x04D8 is Microchip)
         0xff,0x00,           //product id
         0x01,0x00,           //device release number
         0x01,                //index of string description of manufacturer. therefore we point
to string_1 array (see below)
         0x02,                //index of string descriptor of the product
         0x00,                //index of string descriptor of serial number
         USB_NUM_CONFIGURATIONS   //number of possible configurations
   };

////////////////////////////////////////////////////////////////////
///
///   start string descriptors
```

```
///   String 0 is a special language string, and must be defined.
///   You must define the length else get_next_string_character() will not see the string
///   Current code only supports 10 strings (0 thru 9)
///
////////////////////////////////////////////////////////////////////

//the offset of the starting location of each string.
//offset[0] is the start of string 0, offset[1] is the start of string 1, etc.
const char USB_STRING_DESC_OFFSET[]={0,4,12};

#define USB_STRING_DESC_COUNT sizeof(USB_STRING_DESC_OFFSET)

char const USB_STRING_DESC[]={
   //string 0 -->serial number
         4, //length of string index
         USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
         0x09,0x04,   //Microsoft Defined for US-English
   //string 1 --> company string
         8, //length of string index
         USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
         'A',0,
         'M',0,
         'C',0,
   //string 2 --> device name
         22, //length of string index
         USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
         'U',0,
         'N',0,
         'A',0,
         'M',0,
         ' ',0,
         'F',0,
         '.',0,
         'I',0,
         '.',0,
         ' ',0
};
#ENDIF
```

6.1.2 Archivo fuente PicUSB.c

```
////////////////////////////////////////////////////////////////////////////
////                          PicUSB.h                                  ////
////                                                                    ////
//// This examples shows how to configure the USB device and its        ////
//// descriptors, and how to send and receive data from the PC wait-    ////
//// first for the correct enumeration.                                 ////
////                                                                    ////
//// Compiled using CCS PCWH 4.018                                      ////
////                                                                    ////
//// Author: Abraham Monrroy Cano     abraham_monrroy@hotmail.com       ////
////                                                                    ////
////////////////////////////////////////////////////////////////////////////
#include <18F4550.h>
#fuses HSPLL,NOWDT,NOPROTECT,LVP,NODEBUG,USBDIV,PLL5,CPUDIV1,VREGEN
#use delay(clock=48000000)
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)


////////////////////////////////////////////////////////////////////////////
//
// CCS Library dynamic defines.  For dynamic configuration of the CCS Library
// for your application several defines need to be made.  See the comments
// at usb.h for more information
//
////////////////////////////////////////////////////////////////////////////
#define USB_HID_DEVICE     FALSE             //deshabilitamos el uso de las directivas HID
#define USB_EP1_TX_ENABLE  USB_ENABLE_BULK   //turn on EP1(EndPoint1) for IN bulk/interrupt
transfers
#define USB_EP1_RX_ENABLE  USB_ENABLE_BULK   //turn on EP1(EndPoint1) for OUT bulk/interrupt
transfers
```

```
#define USB_EP1_TX_SIZE    64               //size to allocate for the tx endpoint 1 buffer
#define USB_EP1_RX_SIZE    8                //size to allocate for the rx endpoint 1 buffer

////////////////////////////////////////////////////////////////////////////
//
// Include the CCS USB Libraries.  See the comments at the top of these
// files for more information
//
////////////////////////////////////////////////////////////////////////////
#include <pic18_usb.h>   //Microchip PIC18Fxx5x Hardware layer for CCS's PIC USB driver
#include <PicUSB.h>      //Header, includes all descriptors
#include <usb.c>         //handles usb setup tokens and get descriptor reports


#define modo      recibe[0]
#define param1    recibe[1]
#define param2    recibe[2]
#define resultado envia[0]


void main(void) {

   int8 recibe[3];
   int8 envia[1];

   output_low(PIN_B6);
   output_high(PIN_B7);

   printf("Inicializando\n");
   usb_init();
   printf("Inicializacion terminada\n");

   usb_task();                      //enable usb interface and interrupts
   printf("Esperando enumeracion\n");
   usb_wait_for_enumeration();      //wait until the device gets configured by the host

   while (TRUE)
   {
      if(usb_enumerated())
      {
         printf("Enumerado:OK\n");
         if (usb_kbhit(1))          //si el endpoint de salida contiene datos del host
         {
            usb_get_packet(1, recibe, 3); //receive 3 bytes from Endpoint 1

            usb_put_packet(1, envia, 1, USB_DTS_TOGGLE); //send 1 byte to endpoint 1
         }
      }
   }
}
```

6.2 Código fuente para el desarrollo del controlador para Windows

Los archivos fuente mostrados a continuación forman parte de un proyecto, y son requeridos para la correcta compilación del controlador

6.2.1 Archivo Device.c

```
#include "ProtoTypes.h"
#include "public.h"

#pragma alloc_text(PAGE, EvtDevicePrepareHardware)
#pragma alloc_text(PAGE, ConfigureUsbInterface)
#pragma alloc_text(PAGE, ConfigureUsbPipes)
#pragma alloc_text(PAGE, EvtDeviceAdd)
#pragma alloc_text(PAGE, CreateQueues)

/*.....................................................................*/
/* the callback function that will be called whenever a new device is added.  */
/* this is for PNP devices.                                            */
/*.....................................................................*/
NTSTATUS EvtDeviceAdd(
    IN WDFDRIVER  Driver,
    IN PWDFDEVICE_INIT  DeviceInit
    )
{
  NTSTATUS status;
  WDFDEVICE device;
  PDEVICE_CONTEXT devCtx = NULL;
  WDF_OBJECT_ATTRIBUTES attributes;
  WDF_PNPPOWER_EVENT_CALLBACKS pnpPowerCallbacks;
  WDF_DEVICE_PNP_CAPABILITIES pnpCapabilities;

  UNREFERENCED_PARAMETER(Driver);

  /*set the callback functions that will be executed on PNP and Power events*/
  WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);
  pnpPowerCallbacks.EvtDevicePrepareHardware = EvtDevicePrepareHardware;
  pnpPowerCallbacks.EvtDeviceD0Entry = EvtDeviceD0Entry;//function tu execute when start or
wakeup
  pnpPowerCallbacks.EvtDeviceD0Exit = EvtDeviceD0Exit;//func to execute when shutdown
  WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);//set above funcs

  WdfDeviceInitSetIoType(DeviceInit, WdfDeviceIoBuffered);

  /*initialize storage for the device context*/
  WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, DEVICE_CONTEXT);

  /*create a device instance once configured.*/
  status = WdfDeviceCreate(&DeviceInit, &attributes, &device);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfDeviceCreate failed with status 0x%08x\n", status));
    return status;
  }

  /*set the PNP capabilities of our device. we don't want an annoying
    popup if the device is pulled out of the USB slot.*/
  WDF_DEVICE_PNP_CAPABILITIES_INIT(&pnpCapabilities);
  pnpCapabilities.Removable = WdfTrue;
  pnpCapabilities.SurpriseRemovalOK = WdfTrue;
  WdfDeviceSetPnpCapabilities(device, &pnpCapabilities);

  devCtx = GetDeviceContext(device);

  status = WdfDeviceCreateDeviceInterface(device, &GUID_DEVINTERFACE_FX2, NULL);
  if(!NT_SUCCESS(status))
  {
```

```
    KdPrint((__DRIVER_NAME
      "WdfDeviceCreateDeviceInterface failed with status 0x%08x\n", status));
    return status;
  }

  return status;
}

/*.....................................................................*/
/* create the different IO queues that will be used by for sending different  */
/* types of requests to our driver.                                    */
/*.....................................................................*/
NTSTATUS CreateQueues(WDFDEVICE Device, PDEVICE_CONTEXT Context)
{
  NTSTATUS status = STATUS_SUCCESS;

  WDF_IO_QUEUE_CONFIG ioQConfig;

  //create the default IO queue. this one will be used for ioctl request entry.
    //this queue is parallel, so as to prevent unnecessary serialization for
    //IO requests that can be handled in parallel.
  WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&ioQConfig,
                                         WdfIoQueueDispatchParallel);
  /*ioQConfig.EvtIoDeviceControl = EvtDeviceIoControlEntry;
  status = WdfIoQueueCreate(Device,
                            &ioQConfig,
                            WDF_NO_OBJECT_ATTRIBUTES,
                            &Context->IoControlEntryQueue);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfIoQueueCreate failed with status 0x%08x\n", status));
    return status;
  }

  //create the IO queue for serialize IO requests. This queue will be filled by
    //the IO control entry handler with the requests that have to be serialized
    //for execution.
  WDF_IO_QUEUE_CONFIG_INIT(&ioQConfig,
                           WdfIoQueueDispatchSequential);
  ioQConfig.EvtIoDeviceControl = EvtDeviceIoControlSerial;
  status = WdfIoQueueCreate(Device,
                            &ioQConfig,
                            WDF_NO_OBJECT_ATTRIBUTES,
                            &Context->IoControlSerialQueue);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfIoQueueCreate failed with status 0x%08x\n", status));
    return status;
  }*/

  /*create the IO queue for write requests*/
  WDF_IO_QUEUE_CONFIG_INIT(&ioQConfig,
                           WdfIoQueueDispatchSequential);
  ioQConfig.EvtIoWrite = EvtDeviceIoWrite;
  status = WdfIoQueueCreate(Device,
                            &ioQConfig,
                            WDF_NO_OBJECT_ATTRIBUTES,
                            &Context->IoWriteQueue);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfIoQueueCreate failed with status 0x%08x\n", status));
    return status;
  }

  status  = WdfDeviceConfigureRequestDispatching(Device,
                                                 Context->IoWriteQueue,
                                                 WdfRequestTypeWrite);
  if(!NT_SUCCESS(status))
```

```
  {
    KdPrint((__DRIVER_NAME
      "WdfDeviceConfigureRequestDispatching failed with status 0x%08x\n",
      status));
    return status;
  }

  /*create the IO queue for read requests*/
  WDF_IO_QUEUE_CONFIG_INIT(&ioQConfig,
                           WdfIoQueueDispatchSequential);
  ioQConfig.EvtIoRead = EvtDeviceIoRead;
  status = WdfIoQueueCreate(Device,
                            &ioQConfig,
                            WDF_NO_OBJECT_ATTRIBUTES,
                            &Context->IoReadQueue);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfIoQueueCreate failed with status 0x%08x\n", status));
    return status;
  }

  status  = WdfDeviceConfigureRequestDispatching(Device,
                                                 Context->IoReadQueue,
                                                 WdfRequestTypeRead);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfDeviceConfigureRequestDispatching failed with status 0x%08x\n",
      status));
    return status;
  }

  return status;
}

/*.....................................................................*/
/* call-back function that will be called by the pnp/power manager after the  */
/* Plug and Play manager has assigned hardware resources to the device and    */
/* after the device has entered its uninitialized D0 state. The framework     */
/* calls the driver's EvtDevicePrepareHardware callback function before       */
/* calling the driver's EvtDeviceD0Entry callback function.                   */
/*.....................................................................*/
NTSTATUS EvtDevicePrepareHardware(
    IN WDFDEVICE    Device,
    IN WDFCMRESLIST ResourceList,
    IN WDFCMRESLIST ResourceListTranslated
    )
{
  NTSTATUS status;
  PDEVICE_CONTEXT devCtx = NULL;
  //WDF_USB_CONTINUOUS_READER_CONFIG interruptConfig;

  UNREFERENCED_PARAMETER(ResourceList);
  UNREFERENCED_PARAMETER(ResourceListTranslated);

  devCtx = GetDeviceContext(Device);

  status = ConfigureUsbInterface(Device, devCtx);//single interface device
  if(!NT_SUCCESS(status))
    return status;

  status = ConfigureUsbPipes(devCtx);//check for bulk endpoints in & out
  if(!NT_SUCCESS(status))
    return status;

  status = InitPowerManagement(Device, devCtx);//automatically check for pm capabilities
  if(!NT_SUCCESS(status))
    return status;

  return status;
```

```
}

/*.....................................................................*/
/* configure the USB interface of our device                           */
/*.....................................................................*/
NTSTATUS ConfigureUsbInterface(WDFDEVICE Device, PDEVICE_CONTEXT DeviceContext)
{
  NTSTATUS status = STATUS_SUCCESS;
  WDF_USB_DEVICE_SELECT_CONFIG_PARAMS usbConfig;

  status = WdfUsbTargetDeviceCreate(Device,
                                    WDF_NO_OBJECT_ATTRIBUTES,
                                    &DeviceContext->UsbDevice);

  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfUsbTargetDeviceCreate failed with status 0x%08x\n", status));
    return status;
  }

  /*initialize the parameters struct so that the device can initialize
    and use a single specified interface.
    this only works if the device has just 1 interface.*/
  WDF_USB_DEVICE_SELECT_CONFIG_PARAMS_INIT_SINGLE_INTERFACE(&usbConfig);

  status = WdfUsbTargetDeviceSelectConfig(DeviceContext->UsbDevice,
                                          WDF_NO_OBJECT_ATTRIBUTES,
                                          &usbConfig);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfUsbTargetDeviceSelectConfig failed with status 0x%08x\n", status));
    return status;
  }

  /*put the USB interface in our device context so that we can use it in
    future calls to our driver.*/
  DeviceContext->UsbInterface =
    usbConfig.Types.SingleInterface.ConfiguredUsbInterface;

  return status;
}

/*.....................................................................*/
/* configure the different IO pipes that are available on our device   */
/*.....................................................................*/
NTSTATUS ConfigureUsbPipes(PDEVICE_CONTEXT DeviceContext)
{
  NTSTATUS status = STATUS_SUCCESS;
  BYTE index = 0;
  WDF_USB_PIPE_INFORMATION pipeConfig;
  WDFUSBPIPE pipe = NULL;

  DeviceContext->UsbBulkInPipe = NULL;
  DeviceContext->UsbBulkOutPipe = NULL;
  WDF_USB_PIPE_INFORMATION_INIT(&pipeConfig);
  do
  {
    pipe = WdfUsbInterfaceGetConfiguredPipe(DeviceContext->UsbInterface,//the single interface
                                          index,//indice del pipe
                                          &pipeConfig);//datos leidos del pipe index
    if(NULL == pipe)
      break;//do until there is no other pipes.

    /*none of our data transfers will have a guarantee that the requested
      data size is a multiple of the packet size.*/
    WdfUsbTargetPipeSetNoMaximumPacketSizeCheck(pipe);


    if(WdfUsbPipeTypeBulk == pipeConfig.PipeType)
```

```
    {
      if(TRUE == WdfUsbTargetPipeIsInEndpoint(pipe))
      {
        DeviceContext->UsbBulkInPipe = pipe;
      }
      else if(TRUE == WdfUsbTargetPipeIsOutEndpoint(pipe))
      {
        DeviceContext->UsbBulkOutPipe = pipe;
      }
    }
       index++;
  } while(NULL != pipe);

  if( (NULL == DeviceContext->UsbBulkInPipe) ||
         (NULL == DeviceContext->UsbBulkOutPipe))
  {
    KdPrint((__DRIVER_NAME
      "Not all expected USB pipes were found.\n"));
    return STATUS_INVALID_PARAMETER;
  }

  return status;
}
```

6.2.2 Archivo DeviceIO.c

```
#include <initguid.h>
#include "ProtoTypes.h"
#include "public.h"

#pragma alloc_text(PAGE, EvtDeviceIoRead)
#pragma alloc_text(PAGE, EvtDeviceIoWrite)
#pragma alloc_text(PAGE, EvtDeviceIoControlEntry)
#pragma alloc_text(PAGE, EvtDeviceIoControlSerial)
#pragma alloc_text(PAGE, EvtIoReadComplete)
#pragma alloc_text(PAGE, EvtIoWriteComplete)
/*.....................................................................*/
/* callback function for handling write requests.                      */
/*.....................................................................*/
VOID
EvtDeviceIoWrite(
    IN WDFQUEUE  Queue,
    IN WDFREQUEST  Request,
    IN size_t  Length
    )
{
  NTSTATUS status = STATUS_SUCCESS;
  PDEVICE_CONTEXT devCtx = NULL;
  WDFMEMORY requestMem;

  devCtx = GetDeviceContext(WdfIoQueueGetDevice(Queue));

  UNREFERENCED_PARAMETER(Length);

  KdPrint((__DRIVER_NAME "Received a write request of %d bytes\n", Length));

  status = WdfRequestRetrieveInputMemory(Request, &requestMem);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfRequestRetrieveInputMemory failed with status 0x%08x\n", status));
    WdfRequestComplete(Request, status);
    return;
  }

  status = WdfUsbTargetPipeFormatRequestForWrite(
                                      devCtx->UsbBulkOutPipe,
```

```
                                    Request,
                                    requestMem,
                                    NULL);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfUsbTargetPipeFormatRequestForWrite failed with status 0x%08x\n", status));
    WdfRequestComplete(Request, status);
    return;
  }
  WdfRequestSetCompletionRoutine(Request,
                                 EvtIoWriteComplete,
                                 devCtx->UsbBulkOutPipe);
  if(FALSE == WdfRequestSend(Request,
                          WdfUsbTargetPipeGetIoTarget(devCtx->UsbBulkOutPipe),
                          NULL))
  {
    KdPrint((__DRIVER_NAME "WdfRequestSend failed with status 0x%08x\n", status));
    status = WdfRequestGetStatus(Request);
  }
  else
    return;

  WdfRequestComplete(Request, status);
}

/*.....................................................................*/
/* callback function for handling read requests                        */
/*.....................................................................*/
VOID
EvtDeviceIoRead(
    IN WDFQUEUE  Queue,
    IN WDFREQUEST  Request,
    IN size_t  Length
    )
{
  NTSTATUS status = STATUS_SUCCESS;
  PDEVICE_CONTEXT devCtx = NULL;
  WDFMEMORY requestMem;

  devCtx = GetDeviceContext(WdfIoQueueGetDevice(Queue));

  UNREFERENCED_PARAMETER(Length);
  KdPrint((__DRIVER_NAME "Received a read request of %d bytes\n", Length));

  status = WdfRequestRetrieveOutputMemory(Request, &requestMem);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfRequestRetrieveOutputMemory failed with status 0x%08x\n", status));
    WdfRequestComplete(Request, status);
    return;
  }

  status = WdfUsbTargetPipeFormatRequestForRead(
                                    devCtx->UsbBulkInPipe,
                                    Request,
                                    requestMem,
                                    NULL);//hacer peticion de lectura, pero no enviar peticion
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfUsbTargetPipeFormatRequestForRead failed with status 0x%08x\n", status));
    WdfRequestComplete(Request, status);
    return;
  }
  WdfRequestSetCompletionRoutine(Request,
                                 EvtIoReadComplete,
                                 devCtx->UsbBulkInPipe);//requerir ejecucion de EvtIoReadComplete
al completar la lectura
  if(FALSE == WdfRequestSend(Request,
```

```
                            WdfUsbTargetPipeGetIoTarget(devCtx->UsbBulkInPipe),
                            NULL))//enviar peticion
  {
    KdPrint((__DRIVER_NAME "WdfRequestSend failed with status 0x%08x\n", status));
    status = WdfRequestGetStatus(Request);
  }
  else
    return;

  WdfRequestComplete(Request, status);//completar la peticion y reportar el estado de esta

}
/*.....................................................................*/
/* callback function for signalling the completion of a read request.  */
/*.....................................................................*/
VOID
EvtIoReadComplete(
    IN WDFREQUEST  Request,
    IN WDFIOTARGET  Target,
    IN PWDF_REQUEST_COMPLETION_PARAMS  Params,
    IN WDFCONTEXT  Context)
{
  PWDF_USB_REQUEST_COMPLETION_PARAMS usbCompletionParams;

  UNREFERENCED_PARAMETER(Context);
  UNREFERENCED_PARAMETER(Target);

  usbCompletionParams = Params->Parameters.Usb.Completion;

  if(NT_SUCCESS(Params->IoStatus.Status))
  {
    KdPrint((__DRIVER_NAME "Completed the read request with %d bytes\n",
          usbCompletionParams->Parameters.PipeRead.Length));
  }
  else
  {
    KdPrint((__DRIVER_NAME "Failed the read request with status 0x%08x\n",
          Params->IoStatus.Status));
  }
  WdfRequestCompleteWithInformation(Request,
                                    Params->IoStatus.Status,
                                    usbCompletionParams->Parameters.PipeRead.Length);
}

/*.....................................................................*/
/* callback function for signalling the completion of a write request. */
/*.....................................................................*/
VOID
EvtIoWriteComplete(
    IN WDFREQUEST  Request,
    IN WDFIOTARGET  Target,
    IN PWDF_REQUEST_COMPLETION_PARAMS  Params,
    IN WDFCONTEXT  Context)
{
  PWDF_USB_REQUEST_COMPLETION_PARAMS usbCompletionParams;

  UNREFERENCED_PARAMETER(Context);
  UNREFERENCED_PARAMETER(Target);

  usbCompletionParams = Params->Parameters.Usb.Completion;

  if(NT_SUCCESS(Params->IoStatus.Status))
  {
    KdPrint((__DRIVER_NAME "Completed the write request with %d bytes\n",
          usbCompletionParams->Parameters.PipeWrite.Length));
  }
  else
  {
    KdPrint((__DRIVER_NAME "Failed the read request with status 0x%08x\n",
          Params->IoStatus.Status));
  }
```

```
  WdfRequestCompleteWithInformation(Request,
                                    Params->IoStatus.Status,
                                    usbCompletionParams->Parameters.PipeWrite.Length);
}
```

6.2.3 Archivo Driver.c

```
#include "ProtoTypes.h"
#pragma alloc_text(INIT, DriverEntry)

/*.....................................................................*/
/* entry point for the driver.                                         */
/*.....................................................................*/
NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING  RegistryPath
    )
{
  WDF_DRIVER_CONFIG config;
  NTSTATUS status;

  KdPrint(("DriverEntry of WDF_Usb\n"));

  WDF_DRIVER_CONFIG_INIT(&config, EvtDeviceAdd);

  status = WdfDriverCreate(
                      DriverObject,
                      RegistryPath,
                      WDF_NO_OBJECT_ATTRIBUTES,
                      &config,
                      WDF_NO_HANDLE);

  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfDriverCreate failed with status 0x%08x\n", status));
  }

  return status;
}
```

6.2.4 Archivo Power.c

```
#include "ProtoTypes.h"

#pragma alloc_text(PAGE, InitPowerManagement)
#pragma alloc_text(PAGE, EvtDeviceD0Entry)
#pragma alloc_text(PAGE, EvtDeviceD0Exit)
#pragma alloc_text(PAGE, PowerName)

/*.....................................................................*/
/* this function is called when the device is either started or woken up.     */
/*.....................................................................*/
NTSTATUS
EvtDeviceD0Entry(
    IN WDFDEVICE  Device,
    IN WDF_POWER_DEVICE_STATE  PreviousState
    )
{
  NTSTATUS status = STATUS_SUCCESS;
  //PDEVICE_CONTEXT devCtx = NULL;

  UNREFERENCED_PARAMETER(Device);
  KdPrint((__DRIVER_NAME "Device D0 Entry. Coming from %s\n",
    PowerName(PreviousState)));
  return status;
}
```

```
/*.....................................................................*/
/* this function is called when the device is powered down.            */
/* the current IO is left pending, because otherwise the continuous interrupt */
/* read IO will also be cancelled, and it would have to be reconfigured.      */
/*.....................................................................*/
NTSTATUS
EvtDeviceD0Exit(
    IN WDFDEVICE  Device,
    IN WDF_POWER_DEVICE_STATE  TargetState
    )
{
  NTSTATUS status = STATUS_SUCCESS;
  PDEVICE_CONTEXT devCtx = NULL;

  devCtx = GetDeviceContext(Device);

  KdPrint((__DRIVER_NAME "Device D0 Exit. Going to %s\n",
    PowerName(TargetState)));
  return status;
}
/*.....................................................................*/
/* initialize the power management functionality of our USB device.    */
/* we want it to support system wake-up with the on-board switch, and  */
/* idle- time sleep that puts the device into a sleeping state if it is */
/* idle for a specified duration.                                      */
/*.....................................................................*/
NTSTATUS
InitPowerManagement(
    IN WDFDEVICE  Device,
    IN PDEVICE_CONTEXT Context)
{
  NTSTATUS status = STATUS_SUCCESS;
  WDF_USB_DEVICE_INFORMATION usbInfo;

  KdPrint((__DRIVER_NAME "Device init power management\n"));

  WDF_USB_DEVICE_INFORMATION_INIT(&usbInfo);
  status = WdfUsbTargetDeviceRetrieveInformation(
                               Context->UsbDevice,
                               &usbInfo);
  if(!NT_SUCCESS(status))
  {
    KdPrint((__DRIVER_NAME
      "WdfUsbTargetDeviceRetrieveInformation failed with status 0x%08x\n",
      status));
    return status;
  }

  KdPrint((__DRIVER_NAME  "Device self powered: %d",
    usbInfo.Traits & WDF_USB_DEVICE_TRAIT_SELF_POWERED ? 1 : 0));
  KdPrint((__DRIVER_NAME  "Device remote wake capable: %d",
    usbInfo.Traits & WDF_USB_DEVICE_TRAIT_REMOTE_WAKE_CAPABLE ? 1 : 0));
  KdPrint((__DRIVER_NAME  "Device high speed: %d",
    usbInfo.Traits & WDF_USB_DEVICE_TRAIT_AT_HIGH_SPEED ? 1 : 0));

  if(usbInfo.Traits & WDF_USB_DEVICE_TRAIT_REMOTE_WAKE_CAPABLE)
  {
    WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS idleSettings;
    WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS wakeSettings;

    WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_INIT(&idleSettings,
                                               IdleUsbSelectiveSuspend);
    idleSettings.IdleTimeout = 10000;
    status = WdfDeviceAssignS0IdleSettings(Device, &idleSettings);
    if(!NT_SUCCESS(status))
    {
      KdPrint((__DRIVER_NAME
        "WdfDeviceAssignS0IdleSettings failed with status 0x%08x\n",
        status));
      return status;
```

```
    }

    WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT(&wakeSettings);
    wakeSettings.DxState = PowerDeviceD2;
    status = WdfDeviceAssignSxWakeSettings(Device, &wakeSettings);
    if(!NT_SUCCESS(status))
    {
      KdPrint((__DRIVER_NAME
        "WdfDeviceAssignSxWakeSettings failed with status 0x%08x\n",
        status));
      return status;
    }
  }

  return status;
}

/*.....................................................................*/
/* for debugging purposes it is nice to be able to print out power state      */
/*  names so that we can see where we are coming from, and where we are       */
/*  going to.                                                                  */
/*.....................................................................*/
LPCSTR PowerName(WDF_POWER_DEVICE_STATE PowerState)
{
  char * name = "";

  switch(PowerState)
  {
  case PowerDeviceUnspecified: name = "PowerDeviceUnspecified"; break;
  case PowerDeviceD0: name = "PowerDeviceD0"; break;
  case PowerDeviceD1: name = "PowerDeviceD1"; break;
  case PowerDeviceD2: name = "PowerDeviceD2"; break;
  case PowerDeviceD3: name = "PowerDeviceD3"; break;
  case PowerDeviceMaximum: name = "PowerDeviceMaximum"; break;
  default:
    name = "PowerDeviceUnknown"; break;
  }
  return name;
}
```

6.2.5 Archivo ProtoTypes.h

```
#ifndef __PROTOTYPES_H
#define __PROTOTYPES_H

/*revert to warning level 3 to prevent problems with the DDK declarations.
  just disabling the warnings does not work because they get set back to default
  inside ntddk.h.
  NOTE: apparently this behavior is corrected in the vista DDK.*/
#pragma warning(push, 3)

#pragma warning(disable:4200)  // nameless struct/union
#pragma warning(disable:4201)  // nameless struct/union
#pragma warning(disable:4115)  // named typedef in parenthesis
#pragma warning(disable:4214)  // bit field types other than int

#include <ntddk.h>
#include "usbdi.h"
#include "wdf.h"
#include "wdfusb.h"

#pragma warning(pop)


#define __DRIVER_NAME         "WDF_USB: "
/*declaration of the device context. we have to declare the type of the
  device context, and an accessor function name that will return to us
  a pointer to the device context.*/
typedef struct _DEVICE_CONTEXT {
```

```
  WDFUSBDEVICE      UsbDevice;
  WDFUSBINTERFACE   UsbInterface;
  WDFQUEUE          IoWriteQueue;
  WDFQUEUE          IoReadQueue;
  WDFQUEUE          IoControlEntryQueue;
  WDFQUEUE          IoControlSerialQueue;
  WDFQUEUE          SwitchChangeRequestQueue;
  WDFUSBPIPE        UsbBulkInPipe;
  WDFUSBPIPE        UsbBulkOutPipe;
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;

WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(DEVICE_CONTEXT, GetDeviceContext);

NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING  RegistryPath);

NTSTATUS
EvtDeviceAdd(
    IN WDFDRIVER        Driver,
    IN PWDFDEVICE_INIT  DeviceInit
    );

NTSTATUS
EvtDevicePrepareHardware(
    IN WDFDEVICE    Device,
    IN WDFCMRESLIST ResourceList,
    IN WDFCMRESLIST ResourceListTranslated
    );

VOID
EvtDeviceIoRead(
    IN WDFQUEUE  Queue,
    IN WDFREQUEST  Request,
    IN size_t  Length
    );

VOID
EvtDeviceIoWrite(
    IN WDFQUEUE  Queue,
    IN WDFREQUEST  Request,
    IN size_t  Length
    );

VOID
EvtDeviceIoControlEntry(
    IN WDFQUEUE  Queue,
    IN WDFREQUEST  Request,
    IN size_t  OutputBufferLength,
    IN size_t  InputBufferLength,
    IN ULONG  IoControlCode
    );

VOID
EvtDeviceIoControlSerial(
    IN WDFQUEUE  Queue,
    IN WDFREQUEST  Request,
    IN size_t  OutputBufferLength,
    IN size_t  InputBufferLength,
    IN ULONG  IoControlCode
    );
NTSTATUS
EvtDeviceD0Entry(
    IN WDFDEVICE  Device,
    IN WDF_POWER_DEVICE_STATE  PreviousState
    );

NTSTATUS
EvtDeviceD0Exit(
    IN WDFDEVICE  Device,
```

```
    IN WDF_POWER_DEVICE_STATE  TargetState
    );
NTSTATUS
InitPowerManagement(
    IN WDFDEVICE  Device,
    IN PDEVICE_CONTEXT Context
    );

VOID
EvtIoReadComplete(
    IN WDFREQUEST  Request,
    IN WDFIOTARGET  Target,
    IN PWDF_REQUEST_COMPLETION_PARAMS  Params,
    IN WDFCONTEXT  Context
    );

VOID
EvtIoWriteComplete(
    IN WDFREQUEST  Request,
    IN WDFIOTARGET  Target,
    IN PWDF_REQUEST_COMPLETION_PARAMS  Params,
    IN WDFCONTEXT  Context
    );

NTSTATUS
CreateQueues(
    WDFDEVICE Device,
    PDEVICE_CONTEXT Context
    );

NTSTATUS
ConfigureUsbInterface(
    WDFDEVICE Device,
    PDEVICE_CONTEXT DeviceContext
    );

NTSTATUS
ConfigureUsbPipes(
    PDEVICE_CONTEXT DeviceContext
    );

LPCSTR
PowerName(
    WDF_POWER_DEVICE_STATE PowerState
    );
#endif
```

6.2.6 Archivo public.h

```
#ifndef __WDF_USB_DD_PUBLIC__
        #define __WDF_USB_DD_PUBLIC__

        /*WDF USB device GUID*/
        // {C6B78DFF-B260-4161-84ED-09CA267F3E15}
        DEFINE_GUID(GUID_DEVINTERFACE_FX2,
        0xc6b78dff, 0xb260, 0x4161, 0x84, 0xed, 0x9, 0xca, 0x26, 0x7f, 0x3e, 0x15);

        /*device IO control codes for the WDF USB driver*/
        #define IOCTL_INDEX             0x800
        #define FILE_DEVICE_USB_FX2     0x65500
#endif //__WDF_USB_DD_PUBLIC__
```

6.2.7 Archivo wdf_usb_man.inf

Cabe señalar que este archivo no es parte del código fuente, pero es una parte importante una vez que se ha compilado el controlador, ya que sin este archivo, que es simplemente un archivo de texto plano, no es posible instalar el dispositivo en el equipo.

```
[Version]
Signature = "$Windows NT$"
Class=Sample
ClassGUID={894A7460-A033-11d2-821E-444553540000}
Provider=%CIT%
;CatalogFile=wdf_usb.cat
DriverVer= 18/06/2007



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Class installation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
[ClassInstall32]
AddReg=ClassInstall32_AddReg                        ;;write registry
CopyFiles=ClassInstall32_CopyFiles                  ;;copy the files

[ClassInstall32_AddReg]
HKR,,,,"Ne0's device drivers"
HKR,,Icon,,101

[ClassInstall32_CopyFiles]


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  standard INF sections
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[Manufacturer]                          ;;top level section for device install
%MFGNAME%=DeviceList                    ;;the name of the section that contains
                                        ;;the list of devices for %MFGNAME%
[DestinationDirs]
DefaultDestDir=10,System32\drivers      ;;%windir%\system32\drivers this is
                                        ;;mandatory for WDM drivers.

ClassInstall32CopyFiles=11              ;;system32 on WinNT platforms. all files
                                        ;;for the classinstall are copied to
                                        ;;this folder by default.

[SourceDisksFiles]                      ;;identify the files that are to be
                                        ;;copied during installation.
wdf_usb.sys=1,objchk_wxp_x86\i386\,

[SourceDisksNames]                      ;;identify the source disks for the
                                        ;;files that are specified in the
                                        ;;SourceDisksFiles section
1=%INST_DISK_NAME%                      ;;we only have 1 installation disk.

[DeviceList]                            ;;Specify the list of devices that are
                                        ;;supported by our driver.
%DEV_DESCRIPTION%=DriverInstall,USB\VID_04D8&PID_0001

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  device installation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
[DriverInstall.ntx86]                   ;;specify that this is the installation
                                        ;;for nt based systems. if no postfix is
                                        ;;specified, it is assumed that the
                                        ;;driver is installed on 9x
DriverVer=18/06/2007,1.0.0.1            ;;driver version
CopyFiles=DriverCopyFiles               ;;specify the section that tells the
                                        ;;installer which files to copy.
```

```
[DriverCopyFiles]
wdf_usb.sys,,,2                         ;;copy hello.sys

[DriverInstall.ntx86.Services]
AddService=wdf_usb,2,DriverService      ;;tell the PNP manager which files to
                                        ;;load

[DriverService]
ServiceType=1                           ;;kernel mode driver
StartType=3                             ;;start on demand
ErrorControl=1                          ;;normal error handling. the device is
                                        ;;not critical for the system.
ServiceBinary=%10%\system32\drivers\wdf_usb.sys
                                        ;;tell the system where the driver file
                                        ;;is located.

[DriverInstall.ntx86.hw]
AddReg=DriverHwAddReg

[DriverHwAddReg]
HKR,,SampleInfo,,"%WDF_USB registry key"

;-------------- WDF Coinstaller installation

[DestinationDirs]
CoInstaller_CopyFiles = 11

[DriverInstall.ntx86.CoInstallers]
AddReg=CoInstaller_AddReg
CopyFiles=CoInstaller_CopyFiles

[CoInstaller_CopyFiles]
WdfCoinstaller01000.dll

[SourceDisksFiles]
WdfCoinstaller01000.dll=1 ; make sure the number matches with SourceDisksNames

[CoInstaller_AddReg]
HKR,,CoInstallers32,0x00010000, "WdfCoinstaller01000.dll,WdfCoInstaller"

[DriverInstall.ntx86.Wdf]
KmdfService = wdf_usb, wdf_usb_wdfsect

[wdf_usb_wdfsect]
KmdfLibraryVersion = 1.0

;------------------------------------------------------------------;


[Strings]
NEO="Abraham Monrroy Cano"
MFGNAME="Abraham Monrrroy Cano"
INSTDISK=" Installation Disc"
DEV_DESCRIPTION="Interfaces Laboratorio UNAM F.I."
INST_DISK_NAME="WDF USB device driver installation disk"
```